inside the Timex Sinclair 2000 computer

a guide to the anatomy of the hardware

jeff naylor & diane rogers

inside the Timex Sinclair 2000 computer

a guide to the anatomy of the hardware

jeff naylor & diane rogers

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street,
London WC2R 3LD

ISBN 0 946408 31 9

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by The Leagrave Press Ltd.

CONTENTS

Contents in detail

PART 2

CHAPTER 7
Introducing the Timex
The development of the 2000: a detailed look at the facilities it has to offer and the hardware structure with which it achieves them.

CHAPTER 8
The Memory Map
How the Timex uses its ROM and RAM memory: the way in which extra space is created in the map by Bank Switching.

CHAPTER 9
The Keyboard
The way that the keyboard is constructed, scanned and decoded. How we can use the keyboard from machine code; plus a demonstration program that shows a method of scanning the keyboard.

CHAPTER 10
Pictures on the Screen
The nature of television pictures and how they are displayed; the method by which the Timex generates the signals, and where it stores the information; how we may manipulate the screen by using machine code; the additional video modes.

CHAPTER 11
Sound
The way that the BEEP port makes sounds and how to use it; its role in the tape routines. The secrets of the sound chip explored with the aid of a program.

CHAPTER 12
The Joysticks
All you need to know to use the joystick ports.

CHAPTER 13
The Only Safe Chunk in a Storm
Why Chunk 3 of the memory map is so important; an explanation of the Function Dispatcher and the Bank Switching routines.

Preface

For the benefit of newcomers to computing this book assumes no knowledge on the part of its readers. All it requires is a desire to learn. More experienced readers curious about the Timex may like to 'dip into' the text a little further on. The greatest benefits will come if you read with your computer close to hand, so that experimenting will come naturally. Don't be afraid to wander on your own either, as the best remembered facts are those you discover for yourself!

We would like to thank Chris, Ian and especially Nathalie, for their help during the preparation of the book; and John Young for inspiration.

Part 1

First Principles

CHAPTER 1
Electronics of the Digital Kind

When you discover that the apparently inscrutable performance of modern computers is achieved by using Digital Electronics, do not be panicked into assuming that this will make them any harder to understand than a conventional electronic device such as a radio. If you have tried to comprehend electronics at some time in the past and struggled to understand such things as sine waves, modulation, and capacitance, then take heart: the theory behind digital techniques requires less in the way of abstract concepts or knowledge of mathematics. It is size (a great deal crammed into a small package) and speed of operation that make modern computers so powerful.

So what is the difference between Digital Electronics and the more conventional kind? Broadly speaking, a digital circuit is only concerned with the presence or absence of electricity; in other words, whether parts of itself are ON or OFF. The exact amount of electricity present, providing it falls within certain limits, is unimportant: conventional circuits tend to be much more precise. For example, in a hi-fi system the quantities of electricity flowing out to the loudspeakers reflect exactly the changing levels of sound that the circuit is trying to imitate. In a badly designed system the levels of electricity may not be controlled accurately enough, resulting in distortion.

Charles Babbage is sometimes known as the father of modern computing because he designed the first 'Analytic Engine' in the 1830s. It was indeed an engine; with plungers, levers and cogs interacting in a visible, tangible way. Comprehending the behaviour of a mechanical device is easier on the brain because there is nothing abstract to understand — pulling this lever operates that plunger because they are attached to each other by a wire which we can see. In order to come to terms with an electronic device, it is common to envisage a picture of electricity behaving like, for instance, water, or even little men rushing around. There is nothing to be ashamed of in using these analogies, but they are not accurate enough to cope with complex models. A circuit consisting of a battery, bulb and switch can be resolved in these terms, but try to apply your analogy to a television set and not only will you fail to grasp how it works, you may begin to doubt the behaviour of water, or even, of little men!

In order to have a picture of the nature of electricity, it is first necessary to appreciate a few simple facts about matter; all of which would take some time to prove, and in the final analysis you would still be taking my word for it, just as I am taking someone else's. Let us, therefore, accept that matter, whether it is gas, plastic or orange juice, is made up of extremely small particles called *atoms*, a simple definition of which is "the smallest particle that can exist". In the past the definition would have gone on to say that atoms are indivisible, but now we know that this is not the case; however, nuclear fission is not the concern of this book.

There are well over a hundred types of atoms, and some are more common than others. Materials which consist of only one type of atom are called *elements*: they include such familiar substances as carbon, iron and oxygen. The atoms of one element differ from those of another by their size and structure, varying from the simple hydrogen atom to heavyweights such as uranium.

Other substances consist of different types of atoms grouped together, either as a simple mixture of elements, but often as the result of the atoms bonding together and forming what is known as a *molecule*. Two atoms of hydrogen, for example, locked together with one atom of oxygen make up a molecule of water. Through a long process of educated guesswork, followed up by experiments to prove their theories true, scientists have established a picture of what makes up an atom and it is in the construction of the different types of atoms that the key to electricity lies. Each atom consists of a nucleus of a number of *protons* and *neutrons*, which is the so-called 'indivisible core', the prising open of which leads to the science of particle physics. Around this core orbit *electrons*. Each type of atom has a different requirement of these to make them ideally balanced. Hydrogen has a meagre one, whilst others can have dozens. However, some atoms can sustain a small imbalance in their construction, and either host additional electrons or relinquish some of their normal quota. As you have probably guessed from their name, it is these electrons that lead to all electrical phenomena from lightning to digital watches.

If we consider the behaviour of electrons in a battery, an environment which is easy to visualize, we can begin to understand how electricity can be made to be useful. Batteries are constructed in such a way as to be a source of electricity by containing two areas, one with a surplus of electrons, the other with a deficiency. Some batteries, such as those found in cars, can be recharged when the imbalance between the two areas has equalled out; others, such as torch cells, contain chemicals which cannot easily be rejuvenated.

The two areas are connected to terminals on the outside of the case. The terminal which leads to the area containing the surplus of electrons

is called the *negative* terminal, and it is marked with a minus sign. At first glance this may seem illogical, but, as it is how the early experimenters labelled their batteries, the tradition is too well established to be tampered with now. This negative terminal is the jumping off point for the electrons which are eager to get across to the other terminal, marked with a plus sign and called *positive*. You are probably aware that one of the units used in the measurement of electricity is the *volt*: this can be thought of as the pressure on the electrons to move away from their present, overcrowded host atoms to find a more welcoming home. Electrons and the nucleus of atoms behave in a similar way to magnets: the north pole of a magnet is attracted to the south pole of another magnet, and vice versa, whilst two north poles have the reverse effect, actually repelling each other. Electrons don't like others to be present, so if an atom is hosting too many electrons they tend to put pressure on each other to go away; and, if the opportunity occurs, the surplus tend to leave. Do not think of voltage as the number of spare electrons; this is an indication of the capacity of the battery, and therefore relates to how long that battery would keep working. Voltage is the force with which an electron is trying to escape.

It is time to look at our first circuit. If you study **Figure 1** you should

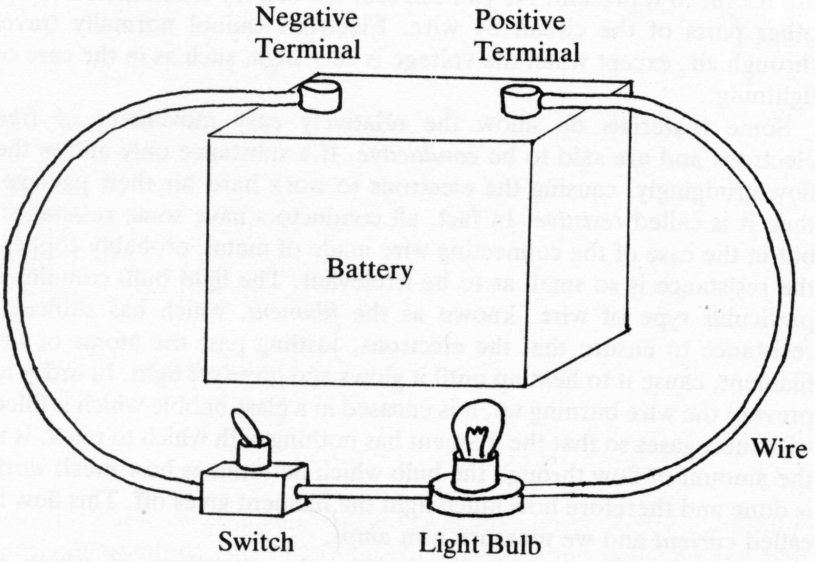


Figure 1

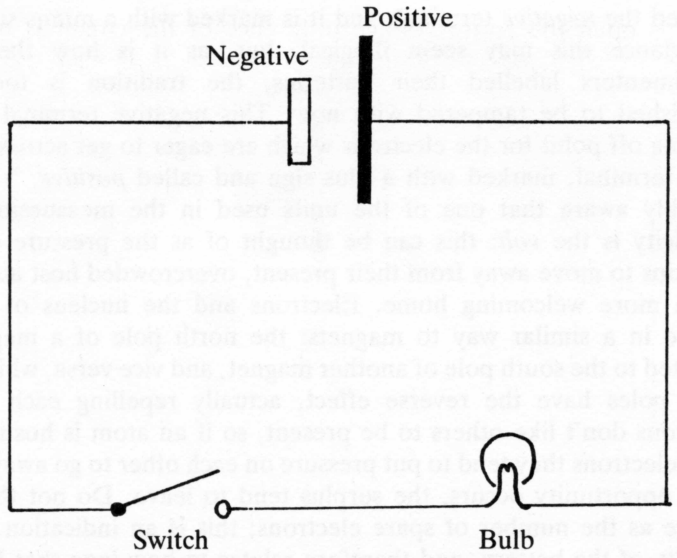


Figure 2

recognize some of the components. **Figure 2** is the same thing drawn as a circuit diagram, using symbols to make the drawing easier and the overall picture simpler to understand once you know what the symbols are meant to represent. As you can see, the battery is connected to the other parts of the circuit by wire. Electrons cannot normally travel through air, except when the voltage is very high, such as in the case of lightning.

Some materials do allow the relatively easy movement of free electrons and are said to be *conductive*. If a substance only allows the flow grudgingly, causing the electrons to work hard for their passage, then it is called *resistive*. In fact, all conductors have some resistance, but in the case of the connecting wire made of metal, probably copper, the resistance is so small as to be irrelevant. The light bulb contains a particular type of wire, known as the *filament*, which has sufficient resistance to ensure that the electrons, jostling past the atoms of the filament, cause it to heat up until it glows and gives off light. In order to prevent the wire burning up, it is encased in a glass bubble which is filled with inert gases so that the filament has nothing with which to react. It is the amount of flow through the bulb which determines how much work is done and therefore how much light the filament gives off. This flow is called *current* and we measure it in *amps*.

The final component is the switch, the operation of which is accurately expressed by its symbol: it provides a mechanical gap in the conductor which can be made good when the switch is closed. When the

switch is open the entire voltage difference of the circuit is present between its two contacts. **Figure 3** shows what happens when the switch

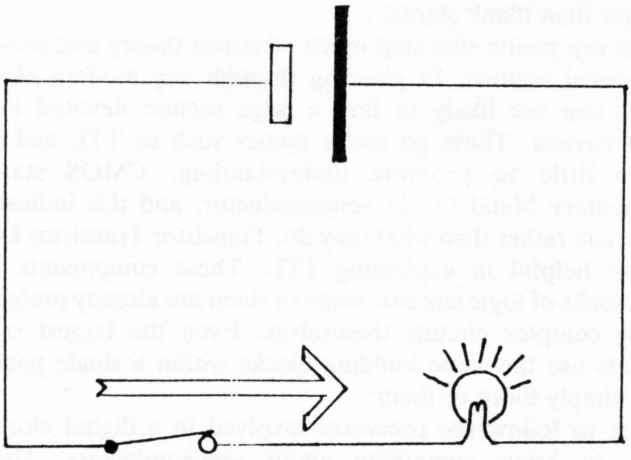


Figure 3

is closed: the voltage present causes electrons to travel along the wire, through the now closed switch, until they reach the bottleneck caused by the resistive nature of the bulb's filament. After passing through the filament the electrons are on the home straight and finally they arrive at their goal, the positive terminal of the battery, where they find an atom which is not overcrowded around which they can orbit.

By now you should have a picture of how the electric circuit in a flashlight behaves. Perhaps you may like to compare it with a simple water analogy, which at this level of complexity holds true. The negative terminal of the battery can be thought of as a tank of water stored in a loft; voltage relates to the height of the tank and therefore to the pressure that is trying to force water through the pipes; the amount of water in the tank is the capacity of the battery. If there is a stopcock it will act like a switch, cutting off the flow; any narrow pieces of pipe would restrict the flow in the same manner as a bulb limits current. You will be able to see that the three parameters of our simple circuit — voltage, current and resistance, are related to each other. Connect a bulb with a higher resistance into the circuit and the flow of current will be smaller; use a battery with a higher voltage and more flow will ensue. Their relationship is fixed in mathematical terms as *Ohm's Law*. This states that the current in amps flowing through a circuit is equal to the voltage in volts, divided by the resistance of the circuit, measured in *ohms*. This means that we can calculate any one of these values if we

know the other two. Nowhere in this book will you be expected to perform such mathematics, but I trust that when I use the words voltage, current and resistance, I can safely visualize comprehending faces rather than blank stares.

Now we can neatly side-step much electrical theory and move on to more practical matters. In glancing through any modern electronics catalogue, you are likely to find a large section devoted to *digital integrated circuits*. These go under names such as TTL and CMOS, which do little to promote understanding. CMOS stands for Complementary Metal Oxide Semiconductor, and this indicates how they are made rather than what they do. Transistor Transistor Logic is a little more helpful in explaining TTL. These components are the building blocks of logic circuits: some of them are already prefabricated into fairly complex circuits themselves. Even the largest computer components use the same building blocks within a single package — there are simply more of them.

In order to follow the processes involved in a digital circuit, it is necessary to know something about *semiconductors*. These are substances which do not occur in nature. They are manufactured from highly refined materials such as germanium and silicon to have particular, very useful properties. The simplest device we can build from semiconducting materials is a *diode*, which passes current in only one direction. Even more useful is the *transistor*, the resistance of which varies when a voltage is applied at a *third* terminal. If we revert to the water analogy, the transistor is like a tap, and the voltage present at the third terminal can be thought of as a hand on the tap, controlling the flow. In analogue circuits the precision of these devices is critical, as the amount of flow allowed through the transistor is proportional to the control voltage. In digital circuits, however the next stage in the circuit is only concerned as to whether there is voltage present or not, so the transistors can be less precise in their manufacture as they are only acting as electrically controlled switches.

I have talked about things being ON or OFF, and I should also mention some other terms. If a positive voltage is present then that part of the circuit is said to be *high*, or, at *logic level one*. Conversely, the negative side of the circuit or supply is said to be *low*, or, at *logic level zero*. There is another possibility, which occurs when we are examining an area not connected to either terminal of the voltage supply, and this is said to be *floating*.

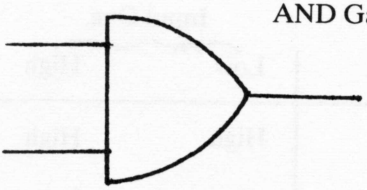
It will be useful to describe in detail the function of one logic chip: the 7409 Quad Two Input AND Gate. Sometimes the names of these parts can indicate what they do; in this case one clue lies in the word Gate. A *logic gate* is a point where a decision is made; but do not assume that any thought is involved: given the same set of circumstances the same gate

will always behave in the same way. The 7409 contains four such gates (hence the Quad in its name), and each gate has two input terminals and one output terminal. It is called an AND gate because it behaves in the following manner. If its first input is high, *and* its second input is also high, then the output will be high. If either, or both inputs are low, then the output will also be low. You may be wondering why there are four gates in one package; simply, the cost is so low that the manufacturers might as well make the most of the space. The chip has 14 pins connecting it to the outside: four sets of three as access to the gates; and two for the voltage supply. The 74 series runs on a 5 volt positive supply, the negative terminal being called 'ground', or GND for short.

A useful technique in the study of logic is the drawing up of a truth table. This is a method of assessing how a logic circuit operates and it is worthwhile making use of a simple one in order to define the operation of an AND gate. Take a look at **Table 1**: across the top of the table are

		Input One	
		Low	High
Input Two	Low	Low	Low
	High	Low	High

Table 1



the possible states of input one of an AND gate, and down the side those of the second input. Follow the 'input one low' row across to where it crosses the 'input two low' column and you read 'low' — the state of the output when both inputs are low. Now study **Table 2**: this describes the behaviour of an OR gate; where the output is high if either input one *or* input two is high. With **Tables 3** and **4** we come across two new words: NAND and NOR. However, these are only AND and OR with an N prefix, which stands for not. If you made a truth table for an AND gate but you lied each time you wrote an answer, the final result would describe the operation of a NAND gate. One way of achieving this electronically would be by using a circuit called an *invertor* attached

Input Two \ Input One	Low	High
Low	Low	High
High	High	High

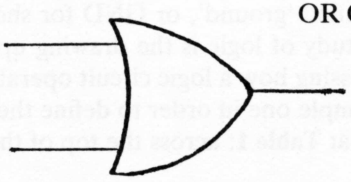


Table 2

to the output of an AND gate. This is one of the simplest logic devices available as it has only one input and output. As its name implies, the output is an inverted reflection of its input; it swops high for low and vice versa. We can also pass the output of an OR gate through an invertor to create a NOR gate.

Input Two \ Input One	Low	High
Low	High	High
High	High	Low

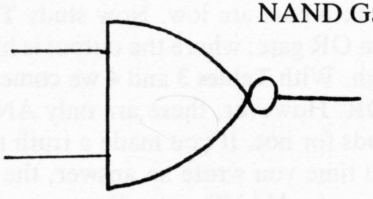
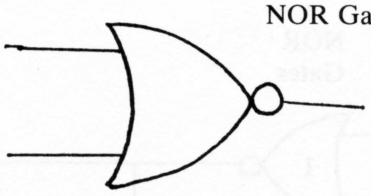


Table 3

		Input One: Low	Input One: High
Input Two	Low	High	Low
	High	Low	Low

Table 4



Now take a good look at **Figure 4**. Spend a little time to work out what will happen when it is turned on, and when the switches are pressed. The circuit represents a latch, or simple flip-flop; and it shows how a pulse, in the form of someone pressing the switch for a moment (ie a logic high being applied briefly to one input of Gate One), achieves a permanent result. The circuit consists of two NOR gates, two switches, and two components that greatly impede the flow of current, namely resistors. Study how the switch and resistor are arranged. If the switch is open there will be zero volts at the point where the wire leading to one input of the gate is connected to the switch and resistor. Close the switch, and current will flow through the switch and resistor so that the point connected to the gate will be at a high voltage.

Let's follow the operation of the gates. When first connected and before either switch is closed, both inputs to Gate One will be at zero volts. Check this occurrence against the NOR gate truth table and you can see that the result will be a high voltage at the output terminal. The inputs to Gate Two will be low in the case of the terminal connected to the switch, and high for the one attached to output one, resulting in a low output at output two. This ties up with the second input to Gate One, keeps it at a low voltage and so maintains the status quo: ie output one high, output two low. Now imagine what happens when you close the switch marked SET. The first NOR gate now has a high and a low on its inputs, resulting in a low output. This affects Gate Two also, as it now has two low inputs, so its output goes high. This has an effect on Gate One, by making both inputs high; but, as you can see from the truth table, its output remains low. We have a new, but stable situation, with the two output values swopped over.

Now let's see what happens when we reopen the SET switch. Gate One has one low and one high input, so the output remains the same. In order to make the outputs flip back we need to press the RESET switch. Try tracing the logic levels that will result. This circuit could be said to remember which switch was pressed last: if it was SET then output one will be high. Extra circuits can be added to make this device work as a store: it can remember a logic level; a property which, as you will see later, is very useful indeed.

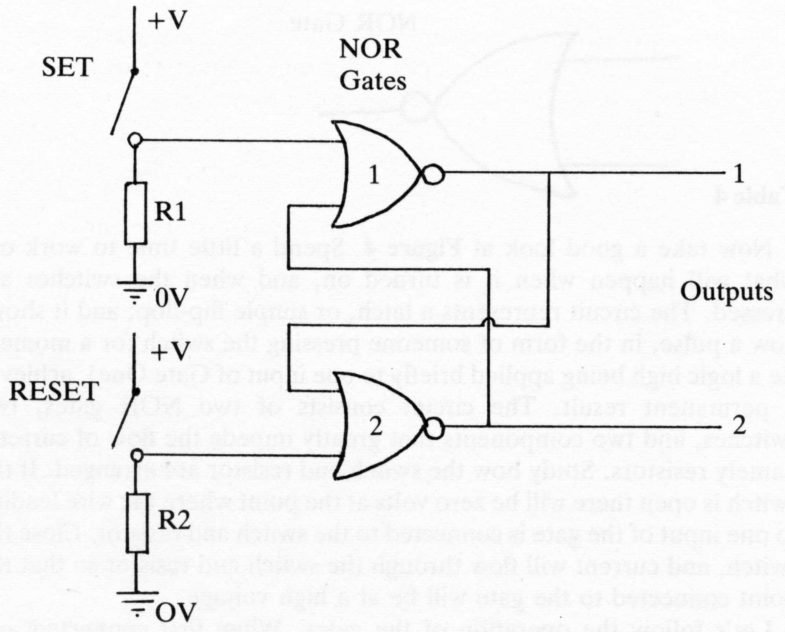


Figure 4 The Flip-Flop

CHAPTER 2
Numbers and Data

In the days of telegraph, communication along the wires was in digital form: with Morse Code using dots and dashes rather than ones and zeros. The process was slow and cumbersome as the information which was to be sent had to be coded, passed on as a series of dots and dashes, and decoded at the other end of the wire. If only something more complex than the on/off signal could be sent, it would all be a lot quicker. For example, by transmitting 26 uniquely differing signals along the wire, the whole alphabet could be represented.

It is at this point that analogue and digital techniques would tackle the problem in different ways. An analogue solution might be to vary the length of the dashes, or to place a voltage of changing levels on the line: continue along this path and you will end up with the telephone. The digital solution, however, is to add more lines: if one line can be at one or zero, then two lines can indicate any one of four conditions: both lines off; line one on and line two off; line one off and line two on; or both lines on. Three lines can offer eight permutations; the four available from two lines with the third line on or off in each situation. Imagine a system of communication which consists of five wires connecting two devices able to code and decode information. You would be able to have 32 buttons, labelled with letters of the alphabet, for your message. A fairly simple digital circuit could then codify this into a unique five-line signal of ons and offs to transmit to the receiver. This would decode the message and it could respond by lighting a particular bulb. This kind of device could, in computer jargon, be called a *Five-Bit Parallel Data Transfer System*: that is, it passes information (data) of five bits (or ones and zeros) along five wires running in parallel.

Let us now consider the same view in a more theoretical way. How people think of numbers depends on their education. I was not introduced to the concept of numbers to a different base until I was too set in my ways to grasp it quickly but modern maths now teaches early on the idea of numbers to bases other than ten. For those puzzled even by the word 'base', an explanation is required. We have ten numbers that only need one figure to represent them: zero to nine. Each time we

reach ten we add to the next column, and so ten is said to be the base. But why only ten figures? If the human race had evolved with 16 fingers and thumbs, we might have had six more single figures to represent the numbers 10 to 15, and then we would have worked to the base 16. Try writing out numbers, say from 1 to 30, using the base 16, with the letters A to F to represent 10 to 15; and compare your results with **Table 5**. You are using hexadecimal numbers, a numbering system often favoured by machine code programmers because it is easier to translate into binary, which is the way that the computer itself handles numbers. As its method of representing data only permits its columns to contain two types of figures, 0 and 1, in order to represent a number greater than 1, the computer needs to use more columns. Instead of units, tens, hundreds, thousands and so on, the columns go up in value by the power of two; ie 2,4,8,16 etc. The TS 2000, together with the majority of home computers, uses a data structure of eight such columns, and can therefore handle numbers up to 255: this is the decimal number which, when it is a binary number, or number to the base of two, is represented as 11111111.

DEC	HEX	DEC	HEX
0	00	16	10
1	01	32	20
2	02	64	40
3	03	128	80
4	04	256	0100
5	05	512	0200
6	06	1024	0400
7	07	2048	0800
8	08	4096	1000
9	09	8192	2000
10	0A	16384	4000
11	0B	32768	8000
12	0C	65535	FFFF
13	0D		
14	0E		
15	0F		

Table 5 Converting decimal to hex

The columns or lines which convey the eight separate bits or voltages around the computer are known collectively as the *data bus*; eight bits of information grouped together to represent a single piece of data are called a *byte*; and 8-bit micros handle one byte of data at a time. They

can store, and communicate with other devices, using numbers in the range 0–255. The value of using hexadecimal (hex for short) is that one byte can be expressed as a two column number in the range 00–FF. With a little practice you should be able to convert mentally a binary number into a hex number and vice versa quite quickly, so that you can visualize how the individual lines of the data bus are behaving without the inconvenience of reading and writing eight column numbers. If you do find hex absolutely impossible, you can get away without using it, but you will still need to become familiar with binary.

I have included a program which can be used either to demonstrate the relationship between binary and decimal, or to calculate the binary form of a decimal number. This program, and all the others in this book, is marked extensively with *remark statements* (REM), which you can omit if you want to save time. Although this first program contains no machine code, later ones will do so and you may lose them completely if you run them with a typing error included. In order to prevent this, I strongly recommend that you save and verify anything that takes more than a minute to enter. As all the programs start from line 10, you can do this using the instruction Save 'name' Line 10: then they will run automatically when reloaded.

Program 1 Binary Numbers

```
si   10 REM    Binary demonstration
  11 REM    program
 100 REM
 101 REM    Set up
 103 REM
 110 GO SUB 400: LET number=0: GO SUB 30
0
 115 REM
 116 REM    Menu
 117 REM
 120 PRINT  AT 20,1;"Press "; INVERSE 1;
"C"; INVERSE 0;"alculate or "; INVERSE 1
;"D"; INVERSE 0;"emonstrate"
 130 LET a$= INKEY$ : IF a$="" THEN  GO
TO 130
 140 IF a$="d" THEN  GO SUB 500: GO TO 1
20
 150 IF a$="c" THEN  GO SUB 600: GO TO 1
20
 160 GO TO 130
 300 REM
```

```
 301 REM   Display Binary Number
 302 REM
 310 LET power=128: LET remain=number
 320 FOR x=0 TO 7
 330 LET result=1: LET remain=remain-pow
er
 340 IF remain<0 THEN  LET remain=remain
+power: LET result=0
 350 LET z=x*4+1: LET color=4-2*result:
FOR y=3 TO 9: PRINT  AT y,z; PAPER color
;"  ": NEXT y
 360 PRINT  AT 12,z;result
 370 LET power=power/2: NEXT x
 380 LET a$="00"+ STR$ number: LET a$=a$
( LEN a$-2 TO  LEN a$): PRINT  AT 17,22;
a$
 390 BEEP .5,number/4: RETURN
 400 REM
 401 REM    Display Screen
 402 REM
 410 BORDER 4: PAPER 6: INK 1: CLS
 420 LET power=256: FOR x=0 TO 7: LET po
wer=power/2: LET z=x*4+1: PRINT  AT 1,z;
"d";7-x; AT 2,z; INK 3;power: PLOT 32*x+
7,96: DRAW 0,55: PLOT 32*x+24,96: DRAW 0
,55: NEXT x
 430 PRINT  AT 0,0;"("; AT 0,31;")": PLO
T 6,174: DRAW 244,0: PRINT  AT 0,12; INK
 7; PAPER 1;"DATA BUS"
 440 PRINT  AT 13,0;"("; AT 13,31;")": P
LOT 6,64: DRAW 244,0: PRINT  AT 14,10; I
NK 7; PAPER 1;"BINARY NUMBER"
 450 PRINT  AT 17,5;"Decimal Number--"
 460 RETURN
 500 REM
 501 REM    Demonstration
 502 REM
 510 PRINT  AT 20,1;"Press "; INVERSE 1;
"M"; INVERSE 0;"enu or hold down "; INVE
RSE 1;"F"; INVERSE 0;"reeze"
 520 FOR a=1 TO 255: LET number=a: GO SU
B 300
 530 IF  INKEY$ ="m" THEN  RETURN
 540 IF  INKEY$  <> "" THEN  GO TO 540
```

```
 550 NEXT a: RETURN
 600 REM
 601 REM     Calculate
 602 REM
 610 PRINT  AT 20,1;"Enter a value from
zero to 255"
 620 INPUT "Decimal Number ? ";number
 630 LET number= INT number: IF number>2
55 OR number<0 THEN  GO TO 620
 640 GO SUB 300: RETURN
```

When Program 1 is entered and safely saved, enter RUN. You will be offered a choice of either a demonstration or a calculation. The calculation mode asks for a number in the range 0–255; then it displays it in two forms. At the top of the screen, there are eight lines representing the data bus of your computer: if these are red then the voltage is high; if green (or the same shade as the border if you are using a monochrome TV), then it is low. Across the screen below the lines is a string of 0s and 1s, showing your chosen number in its binary form. For the demonstration mode, the computer will cycle through the numbers, and you can freeze the display by holding down any key other than the Menu key, M.

So now we have seen how an 8-bit micro deals with data. You may quite rightly argue that your TS 2000 can handle words, and both very large and very small numbers but these are actually handled by the built-in software program. Numbers, for example, are stored and manipulated as five separate bytes of data. Even the machine language which drives the computer sometimes uses a serial form of coding: certain machine instructions come as two or more bytes, with the first giving instructions to the computer to expect further data and how to treat it. These operations will be discussed later. *Parallel Data Transfer* can sometimes extend outside the computer itself: for example, some printers are connected in this manner. When long distances are involved, or speed of communication is less important, it is usual to revert to a serial method. The coding of data into a suitable form may be handled by the computer's host program.

We have looked at the handling of data in a digital form: a similar method is used to direct the flow of that data within the computer. There is another 'bus' inside your TS 2000, called an address bus; it holds a binary number to indicate with which particular area the computer wishes to deal. Each number is unique and can therefore be decoded by using logic circuits to activate the relevant address. (Remember the lighting of the bulb in the 5-Bit Parallel Data Transfer System I mentioned earlier.) In the next chapter you will see how the computer manipulates data by using both the address and data buses.

CHAPTER 3
The Microprocessor

In the not too distant past, computers filled whole rooms: the component parts were housed in separate packages, according to their function. If you had been shown round such an installation, your attention may have been directed towards 'memory' in those big, grey boxes over there, or to 'punched tape reader' here. Your guide may well have indicated one of the anonymous grey cabinets and said: 'that's where the actual work takes place; it's the *Central Processing Unit*'. Computers have now shrunk in size, but if you look inside you can still point to one integrated circuit and say: 'that's where the work is done'. The CPUs, or microprocessors, contain all the components that used to be found in that large, grey cabinet, etched onto a slice of silicon of tiny proportions. Your Timex 2000 series contains a Z80 8-bit microprocessor, a design developed by Zilog in the late 1970s. It is one of the most successful, particularly in the field of small business micro-computers, so I will be using it as a basis for my descriptions. Other microprocessors can be more or less sophisticated, but the general principles are the same. The enclosing of all the processing functions into one circuit encourages us to look at the microprocessor as a kind of 'black box', and just concern ourselves with its inputs and outputs: it is at this point that I intend to start.

A Z80 microprocessor has 40 terminals. Whilst some of these are for inputting, and others for outputting voltages, some of them can perform both functions. To deal with the simplest first, let us study the two terminals (or pins) of the integrated circuit which provide it with its power. For its internal operations and also for outputting to other devices, it needs a supply of five volts positive applied to the pin called the Vcc terminal (supply voltage), relative to the GRN terminal. When this supply is present, a small current passes through the device, the amount of which varies according to the function being performed by the Z80. Next there are the eight data pins, D0 to D7, through which the CPU can pass bytes of data to and from the outside world. These are bi-directional: on some occasions, the eight pins are pulled high or low by the CPU in order to impose a byte of data on the data bus to be used

elsewhere. At other times, the Z80 reads in data placed on the bus by other devices through its data terminals, and when this is happening, these terminals are floating. This type of terminal is very useful and therefore very commonly found on computer chips. It allows you to connect many things to the same data bus without having them interfere with each other, or having large currents flowing along the lines. Chips that are built with such terminals are known as tri-state devices.

Now that we have a method of passing data in and out, we need to control its destination. For this purpose, microprocessors are equipped with address lines which allow them to signal with which one of their surrounding devices they wish to communicate. The Z80 has 16 such address lines. One of the more sophisticated features of this CPU is that it is designed to address two separate forms of peripheral device in a different manner. These are, on the one hand, *memory devices,* which store data, and, on the other, *ports*, which interface with the outside world. Two terminals called IORQ and MREQ indicate with which class of device the CPU wishes to deal. If MREQ is at 0 volts, then the CPU addresses memory; and the address lines hold a 16-bit binary number, revealing with which memory slot the CPU intends to communicate. Eight bits of data give us 256 possible combinations: this is 2 multiplied by itself 8 times which is 2 raised to the power of 8, and can be written as 2^8. 2 raised to the power of 16 comes out as 65,536 and this is normally the maximum number of memory slots (or locations) that the Z80 can address, as 16 is the number of address lines. I say normally because, with additional circuitry and some clever tricks, some microcomputers, the Timex amongst them, are capable of switching between memory devices: this allows for more storage space, and will be described later.

It is worth a short digression in order to explain a piece of computer jargon which can be confusing. Advertisements and computer literature frequently describe a computer as having a certain amount of memory, for instance, 16K bytes. As K is the SI unit for a thousand, (ie the internationally accepted standard), it would be natural to assume that 16K means 16,000 bytes. However, 1K of memory is used to represent 1,024 bytes; this being the amount of memory that can be addressed using 10 bits, (ie 2^{10}). This small discrepancy makes 64K, which is the normal maximum capacity of a Z80 based microcomputer, actually mean 65,536 bytes.

Two further pins called RD and WR (standing for READ and WRITE), tell the memory circuits whether the Z80 requires data from memory, or, on the other hand, whether it is placing a byte on the data bus which it requires to be stored at the location specified by the address bus. These pins are also used when the CPU is addressing a port. We have already seen that the CPU can address memory when the MREQ

is at 0 volts. When IORQ is low, the read and write pins serve the same function they did for memory, applied now to a port; the address of which is held on the eight low address lines. This means that the maximum number of ports that the Z80 can handle is the by now familiar number, 256; and, as with memory, IORQ and the address bus can be decoded to activate the desired circuits. However, it is possible to acquire extra space in the port addressing, as we could do with memory and this will be dealt with under the section devoted to the keyboard.

The essence of a computer system is beginning to emerge, but, in order for the CPU to spring into life, there are two remaining pins to consider. The first is named CLK, which stands for CLOCK: and this is the heartbeat of the Z80. In order to make it function this pin must be alternatively pulled low then high by an external voltage source. This is done by a circuit called an oscillator, which generates a varying voltage in a form known as a square wave. The output of the circuit is high for a fixed period of time and then falls low for the same duration, performing this task continually. If you are interested in seeing why the output is named as it is, enter and run **Program 2**. This plots a graph of voltage

Program 2 Square Wave Graph

```
  10 REM        Square wave graph
  11 REM        plotting program
  12 REM
 100 REM        Set up
 101 REM
 110 PAPER 0: INK 6: BORDER 0: CLS
 120 PRINT "VOLTS = ": FOR x=1 TO 6: PRI
NT  AT x*3,0;6-x: NEXT x
 130 INK 3: PLOT 7,151: DRAW 0,-120: DRA
W 247,0: INK 6
 140 PRINT  AT 1,17;"Square Wave"
 150 DIM a$(9)
 200 REM
 201 REM        Plotting Loop
 202 REM
 210 FOR x=0 TO 400
 211 REM
 220 REM        Calculate volts
 221 REM
 230 LET volts=2.5+10* SIN (x/40)
 240 IF volts>5 THEN  LET volts=5
 250 IF volts<0 THEN  LET volts=0
 300 REM
```

```
301 REM        Plot values
302 REM
310 PLOT  INK 7;8+(24*x/40),32+volts*24
320 REM
321 REM        Print values
322 REM
330 PRINT  AT 0,7;volts;a$
340 PRINT  AT 21,15;x/40;a$
350 REM
351 REM        End loop
352 REM
360 NEXT x
370 STOP
```

against time on the screen of your Timex but the speed of this simulation is very slow compared to the oscillator in your machine. The sequence of the voltage going high, remaining there, going low, and then starting to go high again is called a *cycle;* and we measure the speed of alternating voltages as the number of cycles they perform in one second. The scientist who has this very fundamental unit of measurement named after him is Hertz. Hi-fi buffs will know that the frequency of the sounds that we humans can hear occurs in the range of about 30 Hertz to more than 16 kilohertz, or sixteen thousand cycles per second. The operating speed of the oscillator in your Timex is much higher: 3.5 megahertz (mega stands for million).

Each time the microprocessor senses that the clock input is being pulled from low to high, it performs the next task; so the clock acts as a time keeper for the CPU, regulating its actions and prompting it into doing the next function. With all the complex circuitry that a microprocessor contains, why can it not regulate its own speed, or, even simpler, run as fast as it can? The answer is that the external clock signal gives circuit designers control over the CPU, and allows them to build a computer in which all the associated circuitry can keep pace with the main microprocessor. It also gives them the facility to stop the clock, as it were, to freeze all action until the oscillator is turned on again.

The remaining pin that I must mention is the RESET pin. If you attach it to zero volts it behaves exactly as its name implies it should: it stops the CPU from continuing whatever it is involved with and makes it recommence from a predetermined point.

Let's go through what happens when your Timex, or any other Z80 based microcomputer system for that matter, is turned on. All the circuits begin to receive the voltage they require in order to function. The oscillator circuit starts to apply the square wave clock signal to the CPU but a very simple electronic circuit ensures that, for the first few

moments after the switch on, the reset pin has zero volts applied to it: this causes the processor to start working from the correct place. It is possible to witness what would happen if this was not done, by turning your machine off and on very quickly. If insufficient time is allowed for the reset signal to be generated, the computer may 'hang-up' — it will display a peculiar picture on the screen and fail to respond to the keyboard. When the reset signal is generated correctly, however, it holds the reset pin at zero volts long enough for the CPU to sort itself out. The reset circuitry then allows the appropriate pin to rise to five volts, and the Z80 can start work.

After a reset, the Z80 always does the same thing; it puts the number zero on the address bus, so that all 16 address lines will be set to zero volts by the address pins on the CPU. Then it generates low signals on two other pins, MREQ and RD. Now the Z80 has generated sufficient information to tell its associated circuitry that it requires the data stored in memory location zero to be placed on the data bus, and the memory circuits begin to perform this task. The processor waits for the next clock pulse (the low to high transition of the oscillator circuit); then, it assumes that the data on the data bus is that provided by the memory, and so reads it through its data pins. What the CPU has just fetched from memory location zero is its first machine code instruction, and, still using the clock pulses as a timing reference, the processor goes about the business of performing whatever the instruction tells it to do.

The processes by which the Z80 can store data in memory and exchange information with circuits which are activated by the IORQ pin, should be assumed to be variations of the above description of how it can fetch data from its memory. The time has come to look at what is contained within the CPU itself. This is not as daunting as it may seem, and if you are interested in the nature of machine code, the internal structure of the Z80 is essential knowledge.

CHAPTER 4
Inside the Box

The inside of the Z80 microprocessor can be divided up into a number of different areas: the workings of some of these need not concern the person who wishes to write machine code programs, but an overall view of what goes on can be of great benefit if you want to discover what actually happens when the program is running. The processor contains a number of registers. A register consists of eight flip-flop type circuits each capable of storing one bit of information, connected in such a way so that together they can hold one byte of data: they are in effect the CPU's personal memory cells. The registers are all connected to an internal data bus but note, this is not the same data bus as the one with which the Z80 communicates with the outside world of memory and ports. Whatever appears on these lines is not always transmitted to the data pins, although they are used to carry data to and from the pins when required. Also connected to the internal data bus is circuitry called the Arithmetic and Logic Unit, or ALU: it is in this that all the processing of bytes of data is carried out. Here the CPU can add or subtract two binary numbers, as well as performing logical procedures such as ANDing two bytes together. The whole operation is governed by complex circuitry called the *control unit*, which deciphers the machine code instructions fetched from memory, and reacts accordingly.

Figure 5 merits considerable study: it represents the internal structure of the Z80 processor. The control unit can manipulate all the registers, as well as sending the required signals to the external circuits and the bus control circuits. The instruction register can only be used by the control unit, and its function is to store the last instruction that was fetched from the external memory. All the other registers can be used by the programmer, but some are more versatile than others. The PC register, the program counter, is used to store the *address* of the next instruction to be interpreted: when the CPU is reset, this register is loaded with the value zero, which is the location of the first instruction. When this has been fetched, the control unit automatically adds one to the value in the PC, thus storing the address from which it will receive subsequent data. Unless the programmer uses certain JUMP instructions to reload the PC with a different value, machine code data is always read sequentially. The PC is a 16-bit register, so it can supply any one of 64K addresses to the address bus.

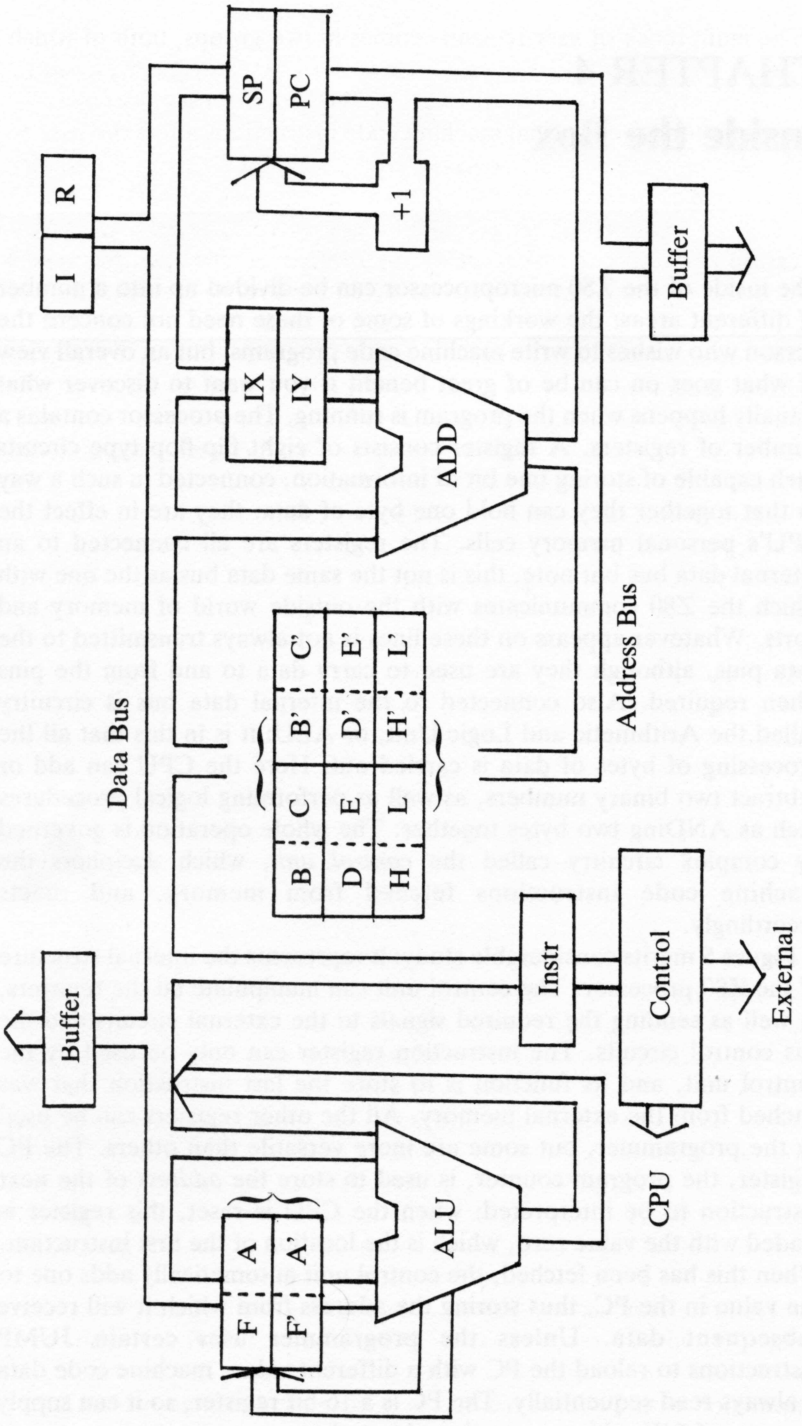


Figure 5 The Central Processing Unit

The main block of user registers comes in two groups, both of which use identical initials as names, but with one set distinguished by a ' suffix to indicate that it is the alternative group. Only one of these sets is active at any one time; and special machine code instructions allow the user to determine which group is connected to the internal data bus in order to receive further instructions. To all intents and purposes, the alternative set can be thought of as useful storage space, particularly if you interrupt work on one task to carry out another: in this case, the values held in the registers can be saved by switching over to the other set for use until you are ready to return to the first task.

The *accumulator*, or A register, is the most versatile of these user registers, because it can have all the arithmetic and logical functions applied to it. The value held in any other user register or stored in external memory can be added, subtracted, ANDed, ORed or XORed with the A register, which will then contain the result.

Certain results of the ALU's calculations can be of particular interest: for instance, when the final product is zero. For its own purposes, as well as for those of the programmer, it notes electronically some of these occurrences, and stores them, as single bits of data, in the F register, sometimes called by its full name, Flags. This conjures up a quaint analogy: imagine a small flag hoisted up a pole marked Z every time a calculation results in zero. Amongst other things, the F register also records if the operation has resulted in a carry; that is, if the answer has exceeded an 8-bit value.

The remaining six 8-bit general purpose registers, B, C, D, E, H and L, can be used as the temporary storage space of values which the CPU can therefore get at quickly; but they can also function as 16-bit registers by pairing up into BC, DE and HL; a facility which is very useful when you are manipulating address values. As pairs, they can have some limited arithmetic operations performed on them, with the result going to the HL register. Another role for the register pairs is as a memory pointer: this means that you can, for example, load the A register with the value held at a memory location, the address of which is held in the HL register.

There exists two special 16-bit registers whose purpose is to hold important memory addresses: important, that is, to the programmer. He is able to load the IX or IY registers with a convenient address, and then have easy and quick access to memory addresses in the range 128 locations below, up to 127 above those held in either of these *index* registers. Two of the less frequently used registers are named I and R. The I, or *interrupt vector*, register, can be used to change the behaviour of the CPU when it receives an interrupt, a subject I will deal with later. The R, or *refresh*, register, is often used if the Z80 is worked in conjunction with dynamic memories: again, I will expand on this later.

The final register for us to consider is the SP, or *stack pointer*, register. This is another form of memory pointer and allows the use of some sophisticated programming techniques. Those of you familiar with BASIC programming will know of the subroutine, whereby the program can be made to jump to a particular line: it remembers where it came from and returns there when it encounters a RETURN instruction. The same principle can be used in Z80 machine code, with the return address stored at the memory location indicated by the stack pointer. This register can have other uses which will feature in the machine code chapter. You should now understand how the CPU exchanges data with the outside world, and have an idea of its internal structure. Let's go slowly through the procedure for executing a very simple program, which we will assume is stored in external memory starting at address zero. Refer to **Table 6** to see the example values.

Address	Contents
0	62
1	8
2	33
3	0
4	88
5	61
6	202
7	15
8	0
9	54
10	0
11	35
12	205
13	5
14	0
15	116

Table 6 Code Example

When the reset pin ceases to be held low by the reset circuit, the Z80, during the first two clock cycles, fetches a byte of data from the memory address held in its program counter. This it passes along its internal data bus to the Instruction register where it is stored for use by the control unit. This data is the first machine code instruction of the program (or OP code, short for operation code): the control unit decodes it and reacts accordingly. In our example program, the first OP

code is 62 decimal, which tells the control unit to fetch the next piece of data stored in memory, and place it in the A register. Two more machine cycles occur whilst this instruction is 'understood': meanwhile, the control unit increases the value in the program counter by one, so that now it holds the value one. Incidentally, the control unit can perform concurrently other tasks related to servicing any dynamic memory circuits which may be attached but this need not concern us.

Back in our example program, the CPU, aware that what is stored at address one is not a further instruction but data, performs another fetching operation from memory and stores it in the A register, as instructed by the previous OP code: it also increments the program counter again. Now, whatever data was stored at address one, in this case 8 decimal, is also stored in the A register. The CPU has completed the task set by the first instruction, so it fetches the next from memory, using the address, currently 2, stored in the PC. The next OP code in the program is dutifully transferred to the instruction register and acted upon. It is 33 decimal, which tells the CPU to put the next two bytes of data into the L and H registers respectively. Now the L register contains whatever was stored at location 3 (0); and H stores the data from location 4 (88 decimal). Remember that one is added to the PC each time a byte is fetched.

We have now reached the instruction in location 5, which is collected in the same manner. This is 61 decimal, which represents the simple task of subtracting one from the contents of the A register. When this is performed, A holds the value 7 decimal. On to address 6, which contains 202 decimal: this is a conditional OP code and could be written as, 'fetch the next two bytes of data from this program and IF the result of the last arithmetic or logic operation was zero, load the data fetched into the program counter'. The CPU is making its first 'decision', based on the contents of the *Flags* register. We made the CPU perform arithmetic only one program step earlier, when it subtracted 1 from the contents of the A register: however, the result was 7 rather than zero. So, although the next two bytes are fetched from addresses 7 and 8, nothing is done with them, as the Z Flag is not set. We will see shortly what happens if it is.

The PC now contains 9, so the CPU loads the contents of this address, which is 54 decimal, into the instruction register: this in effect tells it to 'fetch the next byte from the program as data, and place it in the memory address represented by the contents of the HL register pair'. So the value zero is obtained from address 10, and the control unit goes about the task of storing it in memory. The first step is to place the contents of the L register on the eight low address lines, and the contents of the H register on the eight high address lines: this means that the address bus holds the value 22528 decimal, which is 256 times the

value in H (88), plus the contents of L, in this case zero. Then the control unit pulls the MREQ line low, activating the external memory devices; and it places on the data bus the value it has just fetched from the program memory area, which is zero. It waits a clock cycle for everything to respond and then pulls the WR, or WRITE line low: the external memory takes this as a signal that it should store the data on the data bus, at the address on the address bus. What is the result? The data which was held in memory location 22528, has now been over-written by the data zero.

With the PC now set at 11, the CPU has reached the instruction 35 decimal, which it fetches and reacts to by adding one to the value of the HL register. The data in address 12 is the *unconditional* version of one we have encountered before, at address 6: this time the CPU is told to load the PC with the next two bytes. So, as locations 13 and 14 contain 5 and 0 respectively, these are fetched, and then the PC is loaded with the value 5. (The low part of the PC is loaded with the first byte fetched, and the high part with the second.)

Now where does the next instruction come from? The previous address in the PC has been over-written, so the next instruction comes not from location 15, but from 5. The CPU has performed a JUMP instruction: it loads the OP code from address five, which is 61; an instruction that it has performed before. The program has created a loop: it will continue to perform the OP codes at locations 5 through to 12; but it will not do this indefinitely, due to the conditional instruction at address 6. Each time the loop is carried out, the value of the A register is reduced by one: after the eighth time that the OP code 61 is performed, the A register will have fallen to zero, so when the CPU checks the zero bit of the Flags register, it will find that the previous operation has indeed resulted in zero. This involves proceeding to the second part of the operation dictated by the OP code at address 6: if you refer back you will see that in the instance of the result being zero, the PC is loaded with 15. After executing the loop eight times, the program jumps to the instruction at 15: this happens to be the final one, and it tells the CPU that the program has been completed.

Use **Table 6** to go through each step of the above program carefully until you are clear as to what is happening inside the processor. At this stage it is not important to understand in detail the program itself, but rather to visualize what steps are taken by the internal structure of the Z80 processor, in order to follow through the instructions it fetches from the area of external memory containing the program.

Well, does the program work? And what does it actually do? To answer these questions, why don't you try running it on your Timex? First of all, however, it must be loaded into memory. We cannot load data into the first 16K of memory, as on the Timex this is filled with the

machine's own program, so we will have to use an area higher up in the memory. Consequently, as the first instruction will not be at location zero, we cannot use the reset pin to direct the program counter to it. However, there is a way round this: the Timex has the facility of running machine code from BASIC via the function USR. This is included in **Program 3** in the form of RANDOMIZE USR 28672. It works in the following manner: the address that the BASIC program has reached in its execution is stored at the address indicated by the stack pointer, and the program counter is loaded with the value 28672, so that the CPU starts running the program stored at this address. When the CPU finally encounters the OP code 201 decimal, it reloads the original value of the PC from the area of memory pointed to by the SP: this ensures that the CPU returns to the BASIC program it had been running. BASIC now uses the value that is contained in the BC register of the CPU, and treats it as the result of the function USR. The practical consequence of this is that if we state RAND USR, and then a valid memory address, in any BASIC program we write, the Timex goes to that address and runs any machine code program found there, until it reaches a RET machine code instruction (201 decimal). Then it seeds the random number generator with the value of the BC register. This is just a convenient way of calling up a routine: if we use PRINT USR instead, then the result from BC would be printed on the screen.

Now enter and save **Program 3**. When you run the program, it first of all POKEs the machine code held in the data statements into memory, starting at location 28672. Then it prompts you to press a key, and when you do so it jumps to the program and executes it. A thick black line should appear in the top left hand corner of the screen. Note the speed with which this occurs: what has happened is that the machine code has POKEd zero values in seven memory locations, 22528 to 22535. I chose these particularly because they make up the first part of what is called the *attributes file*: this controls what colours appear on the screen, and where; and it is explained in the section of the book devoted to screen display. You may wish to try changing the value of the A register (the second number in the data statement); the value loaded into the attributes file (eleventh in the data statement); or even the memory pointer in HL itself (but you will need to delete line 60 before other values will work). If you make HL lower than 16348 you will not see the result: any higher than 23296 and not only will nothing show, but you may cause a 'crash' by interfering with the running of the computer. This will not cause any damage, but you may have to switch the machine off and on again to regain control. Incidentally, this machine code program is not only very simple, it is also quite crude; but it does prove that we can bypass the Timex's operating system and exploit the power of machine code programming.

Program 3 Code Demo

ADDRESS	HEX CODE	ASSEMBLER
7000	3E08	LD A, 08h
7002	210058	LD HL, 5800h
7005	3D	DEC A
7006	CA0F70	JP Z, 700Fh
7009	3600	LD (HL), 00h
700B	23	INC HL
700C	C30570	JP 7005h
700F	C9	RET

```
  10 REM      Machine code demo
  11 REM
  20 REM          Set up
  21 REM
  30 PAPER 7: INK 0: BORDER 7: CLS
  40 REM
  41 REM          Check data
  42 REM
  50 RESTORE : LET sum=0: FOR x=0 TO 15:
 READ a: LET sum=sum+a: NEXT x
  60 IF sum <> 1183 THEN  PRINT "The dat
a statements are wrong."''"Please check
that you have"'"entered them correctly":
 STOP
  70 REM
  71 REM    Poke code into memory
  72 REM
  80 RESTORE : FOR x=28672 TO 28687: REA
D byte: POKE x,byte: NEXT x
  90 REM
  91 REM          Wait for key
  92 REM
 100 PRINT  AT 16,10;"PRESS A KEY"
 110 IF  INKEY$ ="" THEN  GO TO 110
 120 REM
 121 REM          Jump to routine
 122 REM
 130 RANDOMIZE  USR 28672
 140 REM
```

```
141 REM        Returned
142 REM
150 PRINT  AT 16,8;"Program complete"
160 STOP
500 REM
501 REM        Machine code
502 REM
510 DATA 62,8,33,0,88,61,202,15
520 DATA 112,54,0,35,195,5,112,201
```

CHAPTER 5
RAM and ROM

In previous chapters, I have glibly mentioned 'external memory' without any further illumination; however, now that we have covered the main working processes of the Z80, it is time for a description. Memory is used by your Timex to store the data it needs in order to function; whether that data is the machine code it uses when it is switched on, the simple programs from this book, or your latest copy of Space Invaders.

There are essentially two distinct types of memory used in microcomputers. Read Only Memory (ROM) does not allow the microprocessor to alter its data: as its name implies, it can only be *read* from. The other is called Random Access Memory (RAM): this is a slightly confusing title because its main feature is the facility for having its contents altered under the control of the computer.

Of course, the situation is not so simple as this sounds: within both RAM and ROM there are different types of memory devices. Let us start by looking at the type of ROM that your Timex uses to store the built-in machine code program which makes it function. There are two Read Only integrated circuits inside your computer: they are essentially the same, although one has double the capacity. This main ROM has 14 address bus pins, labelled A0 to A13.When the Chip Select (CS) pin of the memory is pulled low, the contents of an 8-bit register inside the chip corresponding to the address currently present on the address pins, are placed on the data pins of the memory device (and consequently on the data bus of the computer). As each different address is capable of accessing a separate 8-bit storage area, with 14-bit binary addresses, the ROM is capable of storing 16,484 (16K) different bytes of data. This is actually present in the form of simple connections made at the time of manufacture.

Imagine that the address is decoded by logic circuits in such a manner so as to force the output of one, and only one, logic gate high. If that address were designed to store the number 255, or 11111111 binary, then that gate's output would be connected to all the eight data pins of the ROM. If the data stored was supposed to be 129, which is 10000001 in binary, then the high voltage on the output of the gate would only be

connected to data pins 0 and 7. The data lines are normally low, but the logic gates can pull them high so that the pins will reflect the data stored at an address pointed to by the information on the address bus. When a logic gate is not selected by its appropriate address, its output is floating, and therefore it has no effect on the data bus. When the CS pin is not activated by a logic 0 voltage level, then the whole output of the chip is floating. You could calculate that, if every address held the data 255, then over 130,000 connections would need to be made. This is achieved by the manufacturers using a photographic process to etch the connections onto the silicon material of the chip itself. The template used in this process is very expensive, but once set up the cost of producing each individual chip is low, so these Mask Etched Read Only Memories are economical only for large production runs.

It is worth mentioning two other types of ROM that you may encounter. The *Programmable Read Only Memory* is a device that comes off the production line completely blank: it is programmed by a machine generating high voltages which 'blow' the linking connectors in much the same way a fuse might behave if too great a voltage were applied to it. An improved version, the *Erasable Programmable Read Only Memory,* stores data as electric charges; not only can it be programmed, but also, if necessary, the data can be erased by the application of ultra-violet light through a small window on its upper surface. Both of the above memory device types are used mainly for small production runs and prototypes; and they are often *pin compatible* with normal ROMs. This simply means that they will fit the same sockets and have their pins in the same places. The early prototype Timex 2000 series computers were fitted with EPROMs. They could then be fully tested before the program contained in the EPROMs was committed to an expensive tooling up procedure, to produce the ROM which replaced them.

The most important aspect to notice about all ROMs is that they will retain the data they contain at all times, whether they are powered up or not. Although they need a supply voltage to operate the decoding circuits they contain, the data is stored in a non-volatile form. This is an essential facility for any computer that is to be turned on and off, so that when it is first switched on, there are always some instructions ready and waiting.

Memory which can contain only data that has been preprogrammed is of no use when we want to utilize external memory to store data that the computer can change. Even the machine's own operating system, which controls the keyboard and screen, and interprets the programs that are typed into it, needs more registers than the Z80 can supply. It must remember, for instance, where it last printed something on the screen. So computers are supplied with Random Access Memory, with which

they can write data to locations for later retrieval. The name refers to the fact that locations can be accessed for either reading or writing in any order. (Note that the term Random Access could be applied to the types of ROM I have mentioned; but it has come to be used only for the forms of memory that can be written to.)

The simplest type of RAM can be thought of as a ROM layout with address lines and decoding circuits, but, instead of the links, each bit of information has a flip-flop arrangement whose output represents each *bit* of data. This output is routed to the appropriate data line when the correct address is fed onto the address bus. Also, the data bus is bi-directional, so, with the aid of two pins to tell the chip whether to output or input information, the CPU can send data along with an address to the memory. It will set the values of the appropriate flip-flops to those of the data sent. This kind of memory is called *static RAM,* because as long as it is receiving a supply voltage it will maintain the pattern of bits that its internal circuitry is storing. However, this type of memory is expensive and difficult to make compact, so it will come as no surprise to you to learn that the Timex is equipped with a different breed, called *dynamic memory.* This stores data in the form of an electric charge which, unfortunately, begins to leak away after a short period of time. So the memory needs the opportunity to refresh the charges by receiving a request to read each bit before their values are lost. As I hinted earlier, the Z80 has been equipped to provide refresh signals during the two clock cycles that occur after a memory fetch operation has been performed. The R register is used to provide sequential addresses for the memory circuits to use in conjunction with the refresh pin. From a user's point of view, there is no difference in the performance of dynamic or static Random Access Memories.

My main task in this chapter has been to differentiate between RAM and ROM. The more sceptical amongst you may wish to demonstrate the differences by following this procedure; find the contents of a ROM address, say, location zero, by entering PRINT PEEK 0. Now try to alter the value stored there to, for instance, 1, by entering POKE 0,1. Get the computer to PRINT PEEK 0 again, and you will discover that your efforts have been in vain. Repeat the above operation for the first RAM address, which is 16384: ie, PRINT PEEK 16384; POKE 16384, 1; PRINT PEEK, 16384: Proof! Incidentally, if you have a 16K Timex, all addresses above 32768 will not react, as there are no memory locations in that area.

CHAPTER 6
Language for Machines

It is no use sitting down in front of your computer and typing 'Play me a cheerful tune' — it simply won't know what you're talking about. You can write a program that will play any tune you like, but first of all you must learn how to make the computer understand you. In order to bridge the gap between machine code and human language there has been developed over the years many intermediate languages that can be entered into a computer and understood by both parties. BASIC, the language that your Timex understands, is short for Beginners' All-purpose Symbolic Instruction Code and it is easier to grasp than any other computer language. Although 'professionals' are somewhat disparaging about it, it is flexible and universally accepted as the most popular language. You can if you wish buy other languages that are more elegant to use and faster to run on your machine, but, unless you're a glutton for punishment, I suggest you stick with BASIC for now. I trust that you are familiar with BASIC because it is not within the the scope of this book to explore the finer points of writing programs. If you have yet to write your own BASIC, you will find that the Timex's manual explains this well. In addition there are shelves full of books exploring programming techniques in your local computer store: before we proceed, you should teach yourself BASIC.

So how does BASIC work? The Z80 only understands what the bytes of data fed into it from memory are telling it to do; but not statements such as PRINT AT 2,5; 'HELLO'. Imagine that you have written two programs to beep out on the speaker cheerful and marching music: organize them as subroutines and you can type, for instance, GOSUB 100 for 'Imperial Echoes'. If at the top of the program, you write something such as, INPUT A$: IF A$ = 'Play me a cheerful tune' THEN GOSUB 100: STOP; then when you run the program and type in 'Play me a cheerful tune', your computer will appear to understand you! By adding further lines, perhaps, 'Play me a tune to march to', you may even fool the lay observer into thinking that your machine has musical taste: however you and I know that your micro is interpreting your instructions not through any thoughtful process, but rather by simply comparing them with a list it holds, and responding if your demand tallies with one it recognizes. The machine code program in your Timex

contains a BASIC *interpreter* that works along similar lines. When running, it uses the values of the *keywords* it encounters to find where the machine code needed to perform that task is stored in ROM; then it GOSUBs that routine. (In machine code the word CALL is use instead of GOSUB.) Before moving on to the rather daunting subject of machine code itself, it may help if I explain what goes on when we switch on a Timex and type in 'PRINT AT 2,5; "Hello"' and then press ENTER.

Switch on: the Z80 performs a reset: then it executes the OP codes stored, starting from location zero. These are very involved: setting up the areas of memory, organizing the *system variables* and generally putting the house in order. When the Z80 has finished this *system initialization* routine, it jumps to the *main operating system* loop which waits patiently for someone to press a key. When you do, it knows this to be either a line number or a keyword: by pressing P the operating system looks up how to spell PRINT and places the letters on the screen. It also stores the relevant TOKEN value in an area of memory set aside called the *edit buffer*: each BASIC word has its own token value, the one for PRINT is 245 decimal.

The operating system then follows the same procedure for the AT. Now we enter 2,5;: even more work is involved here, as it stores the relevant numbers as six bytes of data. (This is actually not necessary for small numbers such as 2 and 5, but it is required for larger ones.) The punctuation and string of letters between the quotation marks go into the buffer in the form of their ASCII codes: (you can look these up in your manual.) When you finally press ENTER, the operating system places an ENTER code at the end of the buffer; then it checks the syntax of what has been entered. If nothing is wrong, that is, there is no illegal combination of codes in the line, it moves on to the next stage; otherwise it queries the line. All being well it checks if the data in the buffer starts with a line number. If it had done, the operating system would transfer the string of data to an area of memory set aside to store BASIC programs for running later. As the line you typed in is a direct command, the operating system passes control to the BASIC interpreter which looks up the first value: 245. It recognizes this as a command to pass control to the *output* routine at location 10h; this routine handles the ensuing data by putting it on the screen, dealing with the AT 2,5; tokens by moving the printing co-ordinates accordingly. When the ENTER token is reached, the output routine returns control to the interpreter; this finding the task complete, prints O.K. 0.1., and causes the Z80 to jump back to the Main Operating loop. This example is very simple, but it demonstrates the importance of the system software: without the Operating System (OS) and BASIC language interpreter, the micro would be useless.

Now for the enigmatic 'machine code'. Some readers may already have become familiar with Z80 language, and they will acknowledge that there is great satisfaction in writing a working program. The drawbacks are the number of instructions needed to do anything significant, and the apparently abstract nature of machine code: however there are mnemonics to help us remember the function of each code. If you refer to the listing in Chapter 4, you will see that 62 decimal means 'load A with the next number', which becomes LD A,N. These mnemonics are often called by the overall title *assembly language*, because there are programs available which will translate the names into the correct codes, and thereby assemble a machine code program automatically. In order to assist aspiring machine code users, the next section is devoted to expanding on the more common assembler mnemonics.

Machine Code Mnemonics Glossary

ADC This instruction can apply to either the A or the HL registers. Applied to A it means: 'add the contents of the specified single byte, and the contents of the carry flag, to A and leave the result in A'. Bytes held in single byte GP registers, memory pointed to by HL or the index registers, or stored as immediate data in the program can be added. Applied to HL, the contents of the register pairs (BC, DE, HL) or the stack pointer are added to HL, along with an extra 1 if the carry is set; with the result remaining in HL. For example, ADC A,D adds D to A together with the carry bit.

ADD This is the same as ADC without including the carry bit: in addition to A and HL, it can also be used on the index registers.

AND This is a logic operation that can be performed on the A register. It takes a byte from memory (pointed to by HL or the index registers), program memory or a single GP register, and executes an AND logic test on each bit of A and the corresponding bit of the specified byte, leaving the result in A. For example, AND B would AND the contents of B with A. If A contained 01010101 and B 00001111 binary, the result in A would be 00000101.

BIT By using this instruction you can test any bit from the GP registers, and from memory pointed to by either HL or the index registers and set the zero flag accordingly. For instance, BIT 3, (HL) would test bit 3 of the byte held at address HL and if it were zero the Z flag would be set; if 1 then Z would be

reset. Bits are numbered from 0 to 7. This code is at least two bytes long the first byte always being EDh.

CALL The equivalent of GOSUB in BASIC, CALL pushes the contents of the PC onto the stack, and then loads PC with the next two bytes in program memory, causing the Z80 to execute from that address. See RET to discover how to get back! CALL can be conditional: for example, CALL Z only actually occurs if the zero flag is set.

CCF This stands for COMPLEMENT CARRY FLAG. The value of the carry flag is inverted, zero becomes one and vice versa.

CP This stands for COMPARE: bytes from a particular location in memory, the next program location, or the GP registers, are subtracted from the value in the accumulator, and the flags set accordingly; but the original value of A is restored. If A held 20h, then CP 20 would set the zero flag, but A would remain at 20.

CPD This is a complex instruction. It stands for COMPARE AND DECREMENT, which is useful for searching a block of memory for a byte to tally with that in the A register. CPD compares the byte pointed to by HL with A, and sets the zero and SIGN flags accordingly: (sign = 1 if bit 7 of the result is high.) Then the instruction decrements HL and BC, which acts as a counter and if it reaches zero the P/V (in this case OVERFLOW) flag is set.

CPDR COMPARE AND DECREMENT WITH REPEAT. This will continue to perform CPD until either BC has reached zero or the CP operation has set the Z flag.

CPI COMPARE AND INCREMENT. This is the same as CPD except that HL has 1 added to it rather than subtracted.

CPIR COMPARE AND INCREMENT WITH REPEAT: as CPDR except that it increments HL. These block searching instructions are very useful for finding data of a particular value: on completion, HL will either hold the address of the first match, or the end of the block if none was found.

CPL This stands for COMPLEMENT and applies to the A register only. All the bits are inverted, thus 00110101 would become 11001010.

DEC DECREMENT subtracts 1 from the value it is applied to. You may DEC any 8 or 16-bit register, or a byte in memory pointed to by HL, or the index registers. One important point which, if not known can lead to a frustrating end to first programming attempts, is that decrementing the 16-bit registers (HL, DE, BC,) will not affect the flags register.

DI DISABLE INTERRUPT: this causes the CPU to ignore any MASKABLE INTERRUPT it may receive at its INT pin. Interrupts are covered elsewhere in this book.

DJNZ This useful code means DECREMENT AND JUMP IF NOT ZERO. It enables us to use the B register as a counter: it reduces B by 1, and IF the result does not equal zero, performs a RELATIVE JUMP (see JR), dictated by the next byte from program memory. If zero has been reached then the operation proceeds to the next instruction.

EI ENABLE INTERRUPT: this cancels the DI instruction.

EX This enables you to EXCHANGE two indicated registers. You may EX DE,HL, which swops their contents; you may EX the two bytes pointed to by the SP with those in HL or the index registers: or, EX AF,AF' will switch control to the alternative AF pair.

EXX This command brings the alternative registers BC', DE' and HL' into play, after which any operations are directed towards them. In order to revert to the original set, simply EXX again.

IN This reads a byte from external circuits that are PORT ADDRESSED. It does so by placing the port address on the eight low address lines making IORQ and RD active; then by placing the byte collected on the data bus in the specified register. IN takes two forms: IN A, (port), which has the port number contained in the next byte of program memory with A holding the result; and IN reg (C), for which the byte in C is used as the port address, and reg can be any GP register.

INC INCREMENT adds one to the value held in a specified register: the same rules apply as for DEC.

JP JUMP allows the program counter to be loaded with a new value, which results in the operation of the program moving

elsewhere. There are three alternatives: you may jump to the address indicated in the next two program bytes: perform a conditional JP, which will only occur if a flag test holds true; (ie JP NC, 6000h will load the PC with 6000h, but only if the carry flag is not set): or do an indexed jump which loads the PC with the value held by HL, IX, or IY.

JR JUMP RELATIVE allows you to modify the contents of the PC by adding or subtracting a seven-bit value to the low byte of the PC. The required value is stored in the next byte of program memory. If the byte contains *less* than 127, it is added to the value of the PC; but only after the automatic increment following the fetching procedure: therefore, JR 0 would have no effect; whilst JR 1 would make the CPU omit a byte before fetching its next OP code. If bit 7 of the *displacement byte* is 1; in other words, if the number is greater than 127, this value is treated by a convention known as the Two's Complement, which allows us to generate negative numbers. For example, FEh (254 dec) as a two's complement is minus 2. If you complement (see CMP) the displacement byte and add one, you will arrive at a value which is to be subtracted from the PC. Thus JR FEh (254 dec) forces the CPU to jump back two locations where it will encounter the same instruction, thereby creating an endless loop. You can also use JR as a conditional instruction, but it can only test a limited number of flags: JR Z, JR NZ, JR C, and JR NC. However, JR is often preferable to JP because, as well as being one byte shorter in length, its use absolves the programmer from writing code that must reside in a particular area of memory. JP needs to be accompanied by the actual address; whereas JR only modifies the current value in the PC.

LD This is the most common OP code of all: LOAD. First impressions may suggest that you can load anything with everything, but there are some limitations! In its simplest form LD A,B will transfer the value stored in B and place it in A, destroying the byte previously held in A. Let's systematically go through the LOAD codes that are available.

You may load any 8-bit GP register with immediate data: that is, data stored in the next location in program memory. All the 16-bit registers (bar PC) can be loaded with immediate data stored as the next two bytes of program. See later, the note about how 2-byte values are handled by the Z80.

External memory, of which the address is held in the HL or

index registers (IX and IY plus a displacement, to be explained later) can also be loaded with immediate data: ie *LD (HL),FFh*

It is possible to load the contents of any GP 8-bit register with the contents of any other.

The two special 8-bit registers, I and R, can be loaded with the value in A; and A can be loaded with their contents.

You can only transfer 16-bit values from the double registers into the SP: (however, refer to EX.) To transfer the contents of, say, BC to DE you can, of course, use LD D,B then LD E,C

Data pointed to by the address held in HL or the index registers can be loaded into any 8-bit GP register.

The contents of any 8-bit GP register can be loaded into external memory if the address of the location is held in HL or the index registers.

The A register can also use BC and DE as memory pointers; so LD A, (BC) will load the contents of the address held in BC into the accumulator, and LD (DE),A will load the address held by DE with the value of A.

The extra flexibility of A is shown by the instructions LD A, (ADDR) and LD (ADDR),A. Here ADDR is a 2-byte value stored as immediate data in the next two program locations. This is fetched and used as a memory pointer, so LD A, (0000h) will load A with the bytes of data stored at location zero.

Finally, this technique of using immediate data as a memory pointer can be applied to the 16-bit registers. For example, LD (ADDR),BC will store the contents of BC at two locations, indicated by ADDR and ADDR+1.

LD is fairly extensive, but not infinite! For example, it is not possible to do LD (ADDR), (ANOTHER ADDR). This would require something similar to: LD HL, (ANOTHER ADDR), then LD (ADDR),HL. Also note that no load instruction alters the flags.

LDD This is a complex instruction similar to CPD. It causes the CPU to transfer the data held at the address stored in HL to the address stored in DE; then it decrements the BC, DE, and HL registers. If BC becomes zero then the P/V flag is set.

LDDR As LDD, but with automatic repeat if BC is not zero. You can use this OP code to transfer a block of data from one area of memory to another by loading the following: HL with the last address of the block you want to move; DE with the last address of the destination you require to fill; BC with the number of

bytes to be transferred. LDDR will then carry out the move, and the data will now be in both areas of memory you have chosen.

LDR As LDD but the HL and DE registers are incremented instead of decremented.

LDIR The auto-repeating version of LDI. By choosing either LDDR or LDIR you can fill your new area from the 'top down' or the 'bottom up', which is important if blocks overlap.

NOP NO OPERATION. When the CPU fetches this code it does nothing except increment the PC as always. Surprisingly useful, especially during program development.

OR A logic OP code that can take a byte of immediate data, the contents of a GP register, or a value from memory pointed to by HL, IX or IY, and will perform a bit by bit ORing test between the byte and the A register. The result is left in A, and the flags are set accordingly. For example if A contains 00001111 binary, OR 83h (10000011 binary) will have the result 10001111 in A.

OUT The write version of IN. The same rules apply, except that the data is sent from the register to the specified port.

POP This is an OP code associated with the stack, which is the area of external memory reserved for the CPU's own use: it stores the return addresses for CALL instructions there. POP takes the previous two bytes stored in this area and loads them into the specified register pair: (AF, BC, DE, HL, IX, IY). The stack pointer is then incremented twice, leaving it pointing at the next two bytes to be POPed. Stack operations are dealt with later.

PUSH This reverses the POP procedure: it decrements the SP twice, depositing the contents of the specified register pair at the memory addresses created in the process.

RES RESET to zero: the value of any bit of a byte stored in the GP registers or at a memory location pointed to by either HL or the index registers, can be reset to zero. RES 1,A will make bit 1 of the A register become zero.

RET RETURN from a CALL: this is used at the end of a subroutine in order to return to the original routine. RET retrieves the address that was pushed onto the stack by the CALL OP code and places it in the PC, thereby forcing a jump to that location. Take care if you tamper with the stack or the SP during the subroutine, or you may cause the correct address to become irretrievable. This can sometimes be used on purpose, for example to find the value of the PC.

RL This is the first of a set of ROTATE and SHIFT codes that are best described graphically: refer to **Table 7** for a fuller explanation. However, briefly, RL rotates a byte of data left through the carry flag. The highest bit, bit 7, is placed in the carry flag, and bit 6 shuffles over to replace it. All the bits move along one place, and bit 0 is filled with the old value of the carry bit. Unless otherwise stated, all shift and rotate codes can be applied to any GP register or byte in memory pointed to by HL and the index registers.

RLA An exception to the above rule, RLA can only be applied to the A register. It behaves in the same way as RL, except that no flags other than carry are affected. The rotates that end with 'A' are all only one byte long and are quicker to execute than the others.

RLC ROTATE LEFT CIRCULAR is similar to RL; it differs in that bit 7 is copied into bit 0 as well as the carry, the original value of which is lost.

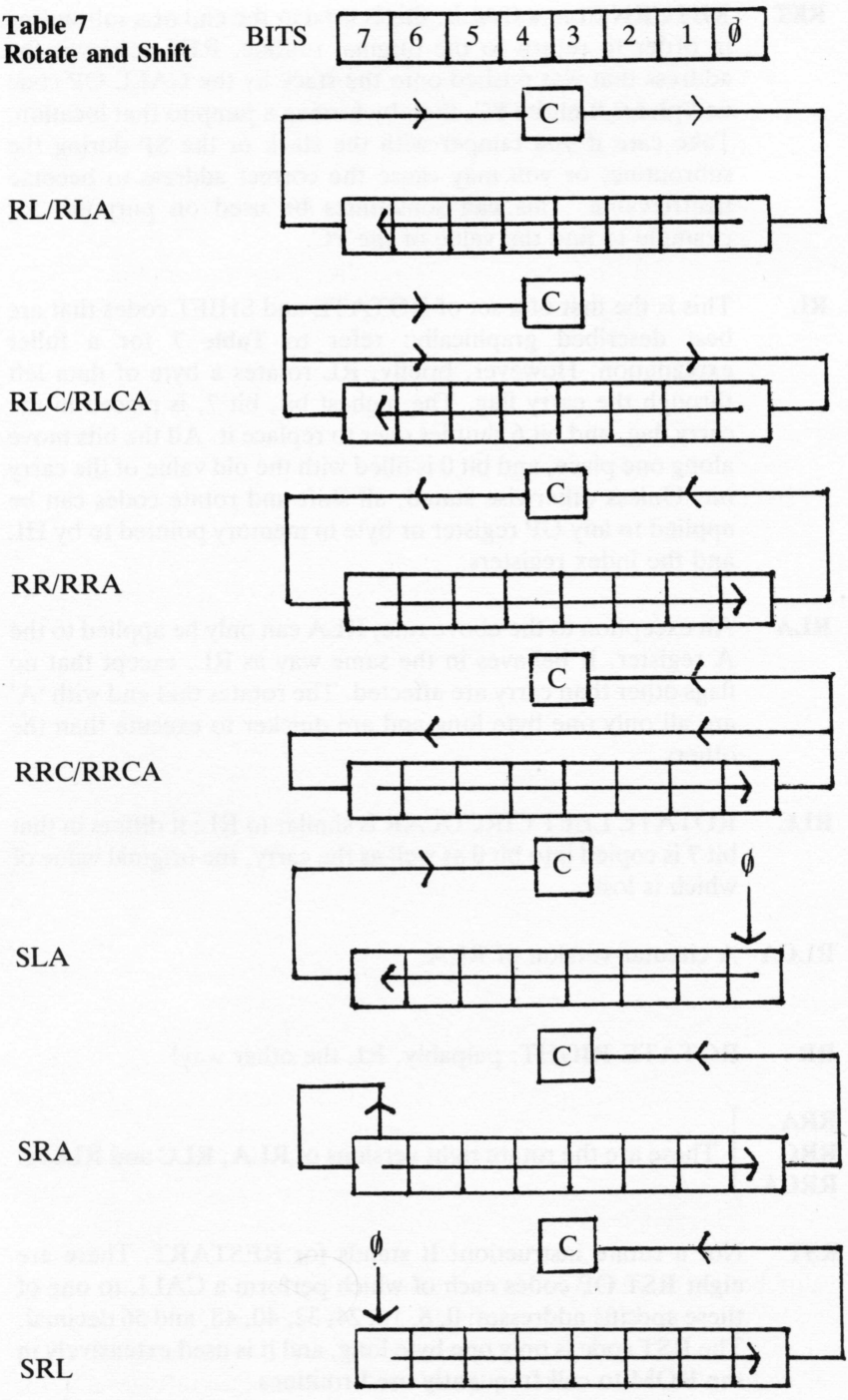
RLCA A circular version of RLA.

RR ROTATE RIGHT: palpably, RL the other way!

RRA
RRC
RRCA } These are the rotate right versions of RLA, RLC and RLCA.

RST Not a rotate instruction! It stands for RESTART. There are eight RST OP codes each of which perform a CALL to one of these specific addresses: 0, 8, 16, 24, 32, 40, 48, and 56 decimal. The RST code is only one byte long, and it is used extensively in the ROM to call frequently used routines.

Table 7
Rotate and Shift



SBC SUBTRACT WITH CARRY. As ADC, except that a subtraction is performed.

SCF SET the CARRY FLAG: this causes the carry flag to become 1, whatever its previous value.

SET See RES. SET makes the specified bit 1.

SLA SHIFT LEFT ARITHMETIC. This is the first of three SHIFT codes. See RL and **Table 7**.

SRA SHIFT RIGHT ARITHMETIC.

SRL SHIFT RIGHT LOGICAL.

SUB This is an 8-bit only OP code that subtracts the contents of either a GP register or external memory pointed to by HL or the index registers from the A register: it leaves the result in A and sets the flags. If you wish to perform a 16-bit SUB then you can use SBC having previously reset the carry flag; AND A is the quickest way to achieve this.

XOR This exotic sounding instruction stands for EXCLUSIVE OR, and is in the same mould as AND and OR. It has a logic one result only if the two bits tested are different from each other: thus both two 0s and two 1s will result in a 0. It will crop up often as XOR A because this is the easiest way to zero the Accumulator.

The Index Registers, Flags and Stack: Notes to accompany the Glossary

The index registers, IX and IY, are often used as memory pointers. OP codes that use them take the form LD A, (IX + DISPLACEMENT BYTE); where displacement byte is a two's complement number (see JR) that is added to the index register, before the value is used as a memory pointer to access a location in memory. Hence, if we load an index register with a suitable number at the start of a program, we can have quick and simple use of a block of memory consisting of 255 bytes centered on a selected value. Timex system software uses the IY register to get at the system variables, so it must always be restored to its correct value before there is a return to the operating system. OP code values can be calculated from the equivalent (HL) code, as DDh is the prefix for IX and FDh for IY. The displacement byte is inserted after the (HL) OP code value; so, in order to transform OR (HL) into OR (IX+2), for

instance, the one byte code B6h would become the three byte code DD B6 02 h.

A word about flags: the CARRY flag is useful for testing to see if one value is greater than another. CP A, 10 will set the carry to 1 if A contains less than 10. The P/V flag used in the block OP codes is normally used to test for overflows when performing two's complement arithmetic, and it reflects the *parity* of the result of a logic operation. An even number of ones in the Accumulator will set the flag. Also remember that you can change the value of flags accidentally with the POP AF and EX AF AF'

One quirk of Z80 code that may confuse you is its method of storing two-byte numbers in external memory: it places the low byte in the first location and the high byte in the next. In the case of LD HL, 0100, it would be stored in memory as 21,00,01. The second byte represents the eight high bits of the value.

Finally, we come to the *stack*. Think of this as a pile of cards: putting a value on stack means that it goes on the top of the pile, so that if we were to pick up, or POP, the top card of value it would always be the last one put down, or the last one exposed when the previous card was taken. The *stack pointer* holds the address of the next card to be POPed. On the Z80, the SP is decremented before a PUSH, so the stack grows 'top down'.

There are some mnemonics missing from the above glossary, but they are not needed for most simple programs and you may think the list complicated enough as it is! If you are interested in writing complex programs then an assembler program and reference book would be a good investment. To give you a taste, however, the above glossary, together with the list of OP codes in the 'Character Set' chapter of the Timex manual will suffice. **Program 4** is a BASIC program that you can use to enter decimal or hexadecimal data into memory; check what you have done; alter, save, and run the machine code you have created. It contains its own operating instructions, so enter it and save it carefully. After running one of your programs it prints to the value left in the BC register, so load that with any result you wish to examine. Try exploring what the various OP codes do, especially the conditional and rotate ones. The size of the program is such that it will just fit a 16K machine, so on that computer you may need to write code in the UDG area.

Now for a challenge! Without looking at **Table 8**, compile a program that fires an arrow across the top line of the screen. You will need to know the following:

1. RST 10h (code D7) will call the ROM's print routine and place the symbol of the code held in the A register on the screen at the current print position.
2. You can change the current print position (it is automatically

updated by the print routine), by sending a 16h (the AT code) in A to the print routine (RST 10h again); then you should follow it with the new print co-ordinates: for examples, LD A, 16h; RST 10h; LD A, 0 RST 10h; will set the print position to the top left hand corner.

3. The code for the arrow can be 3Eh (the 'greater than' sign).
4. You will have to incorporate a 16-bit delay loop if you are to see the arrow travel!

The 'answer' I have provided in **Table 8** is not the sole possibility. You should be able to improve on it when you know the ropes. Have fun!

Program 4 Monitor

```
  1 REM        MONITOR PROGRAM
  2 REM
 10 GO TO 9000
 11 REM
 12 REM        Format 1 byte
 13 REM
 20 IF  NOT h THEN  LET a$="  "+ STR$ a
: LET a$=a$( LEN a$-2 TO ): RETURN
 30 LET a$="  "
 40 FOR x= LEN a$ TO 1 STEP -1: LET b=a
- INT (a/16)*16: LET a$(x)= CHR$ (b+48+7
*(b>9)): LET a= INT (a/16): NEXT x: RETU
RN
 50 REM
 51 REM        Format 2 bytes
 52 REM
 60 IF  NOT h THEN  LET a$="    "+ STR$
a: LET a$=a$( LEN a$-4 TO ): RETURN
 70 LET a$="    ": GO TO 40
100 REM
101 REM     Input n
102 REM
110 LET a$="Invalid number;try again"
120 REM
121 REM     Start here
122 REM
130 LET n=0: INPUT (a$+" "); LINE b$
140 FOR x=1 TO  LEN b$: LET b= CODE b$(
x)-48: IF h THEN  GO TO 170
150 IF b<0 OR b>9 THEN  GO TO 110
160 LET n= INT n+b*10^( LEN b$-x): NEXT
x: RETURN
170 IF b<0 OR b>9 AND b<17 OR b>22 THEN
```

```
 GO TO 110
 180 LET b=b-7*(b>9): LET n= INT n+b*16^
( LEN b$-x): NEXT x: RETURN
 200 REM
 201 REM       Get l
 202 REM
 210 GO SUB 120: IF n<0 OR n>65535 THEN
 LET a$=y$: GO TO 210
 220 RETURN
1000 REM
1001 REM         EXAMINE
1002 REM
1010 CLS : PRINT  INVERSE 1;"M"; INVERSE
 0;"enu,"; INVERSE 1;"N"; INVERSE 0;"ewa
ddress,"; INVERSE 1;"A"; INVERSE 0;"lter
,"; INVERSE 1;"6&7"; INVERSE 0;" scroll"
: LET a$=z$
1020 GO SUB 200: IF n>65516 THEN  LET a$
="Too high;re-enter": GO TO 1020
1030 LET c=0: LET l=n: PRINT  AT 2,0;: F
OR y=0 TO 19: LET p=y: GO SUB 1210: NEXT
 y: GO SUB 1190
1040 LET a$= INKEY$ : IF a$="A" THEN  GO
 TO 1100
1050 IF a$="6" THEN  GO TO 1130
1060 IF a$="7" THEN  GO TO 1160
1070 IF a$="M" THEN  CLS : RETURN
1080 IF a$="N" THEN  GO SUB 1200: LET a$
=z$: GO TO 1020
1090 GO TO 1040
1100 LET a$="New value"
1110 GO SUB 120: IF n>255 THEN  LET a$=y
$: GO TO 1110
1120 POKE l+c,n: LET p=c: GO SUB 1210
1130 IF c <> 19 THEN  GO SUB 1200: LET c
=c+1: GO SUB 1190: GO TO 1040
1140 IF l>65515 THEN  GO TO 1040
1150 LET l=l+1: RANDOMIZE  USR 23335: GO
 SUB 1190: LET p=c: GO SUB 1210: GO TO 1
040
1160 IF c <> 0 THEN  GO SUB 1200: LET c=
c-1: GO SUB 1190: GO TO 1040
1170 IF  NOT l THEN  GO TO 1040
1180 LET l=l-1: RANDOMIZE  USR 23350: LE
```

```
T p=c: GO SUB 1210: GO TO 1040
1190 PRINT  AT c+2,0;"Address>>"; AT c+2
,22;"<<Contents": RETURN
1200 PRINT  AT c+2,0; TAB 9; AT c+2,22,:
 RETURN
1210 LET a=1+p: GO SUB 50: PRINT  AT p+2
,9;a$;"      ";: LET a= PEEK (1+p): GO SU
B 20: PRINT a$: RETURN
2000 REM
2001 REM          ENTER CODE
2002 REM
2010 CLS : PRINT "Mode 2 allows you to e
nter code"; AT 4,2;"ENTER A VALUE OUT OF
 RANGE TO"' TAB 7;"RETURN TO THE MENU";
AT 10,2;"Current address = "
2020 LET a$=z$: GO SUB 200: LET 1=n
2030 LET a=1: GO SUB 50: PRINT  AT 10,20
;a$: LET a$="Code": GO SUB 120: IF n<0 O
R n>255 THEN  RETURN
2040 POKE 1,n: LET 1=1+1*(1<65535): GO T
O 2030
3000 REM
3001 REM          SAVE CODE
3002 REM
3010 CLS : PRINT "Mode 3 will save code
on tape"
3020 LET a$=z$: GO SUB 200: LET 1=n: LET
 a=n: GO SUB 50: PRINT  AT 6,4;z$;" = ";
a$
3030 LET a$="Number of bytes": GO SUB 20
0: LET s=n: LET a=n: GO SUB 50: PRINT  A
T 8,2;"Length of block = ";a$
3040 INPUT "Name of file ";a$: SAVE a$ C
ODE 1,s
3050 PRINT  AT 12,3;"REWIND TAPE AND CHE
CK "' TAB 6;a$;'': VERIFY a$ CODE 1,s
3060 INPUT "O.K. --- Press enter for men
u";a$: RETURN
4000 REM
4001 REM          RUN CODE
4002 REM
4010 CLS : PRINT "Mode 4 will run machin
e code"
4020 LET a$=z$: GO SUB 200: LET a=n: GO
```

```
SUB 50: PRINT  AT 6,3;"The code will be
run from"; AT 8,12;a$; AT 12,3;"Press :-
  R to run code"; AT 14,13;"N to change
address"; AT 16,13;"M for the menu"
4030 LET a$= INKEY$ : IF a$="M" THEN  RE
TURN
4040 IF a$="N" THEN  GO TO 4010
4050 IF a$ <> "R" THEN  GO TO 4030
4060 CLS : PRINT  USR n: INPUT "Code has
 run;press enter";a$: RETURN
5000 REM
5001 REM        CHANGE BASE
5002 REM
5010 CLS : PRINT  AT 3,2;"The Monitor pr
ogram now works"; AT 5,4;"with numbers t
o the base": IF h THEN  GO TO 5030
5020 LET h=1: PRINT  AT 8,12;"SIXTEEN":
GO TO 5040
5030 LET h=0: PRINT  AT 8,14;"TEN":
5040 INPUT "Press ENTER for the menu";a$
: RETURN
6000 REM
6001 REM       CONVERT NUMBER
6002 REM
6010 LET t=h: CLS : PRINT  AT 4,6;"TO CO
NVERT:-"; AT 8,5;"Hex to decimal press H
"; AT 10,5;"Decimal to hex press D"; AT
15,7;"For the menu press M"
6020 LET a$= INKEY$ : IF a$="" THEN  GO
TO 6020
6030 IF a$="M" THEN  LET h=t: RETURN
6040 IF a$="D" THEN  GO TO 6070
6050 IF a$="H" THEN  GO TO 6100
6060 GO TO 6020
6070 LET a$="Decimal": LET h=0
6080 GO SUB 200
6090 LET a=n: LET h=1: GO SUB 60: PRINT
£1;"Hex = ";a$: GO TO 6020
6100 LET a$="Hex": LET h=1
6110 GO SUB 200
6120 PRINT £1;"Decimal = ";n: GO TO 6020
8000 REM
8001 REM       MENU
8002 REM
```

```
8010 CLS : PRINT  AT 1,6;"MACHINE CODE M
ONITOR": PLOT 46,158: DRAW 163,0
8020 PRINT  AT 3,6;"Press the number for
"; AT 4,6;"the desired function"
8030 RESTORE : FOR x=1 TO 6: READ a$: PR
INT  AT x*2+6,5;x;" ----- ";a$: NEXT x
8040 LET a= CODE  INKEY$ -48: IF a<1 OR
a>6 THEN  GO TO 8040
8050 GO SUB a*1000: GO TO 8000
8060 DATA "Examine Memory","Enter code",
"Save Code","Run Code","Change Base","Co
nvert Number"
9000 REM
9001 REM       Set up
9002 REM
9010 INK 7: PAPER 1: BORDER 1: POKE 2365
8,8
9020 LET h=0: LET z$="Start address": LE
T y$="Out of range;try again"
9030 RESTORE 9200: FOR x=23296 TO 23364:
 READ a: POKE x,a: NEXT x
9100 GO TO 8000
9200 DATA 245,230,24,246,64,103,241,245,
230,7,15,15,15,198,9,111,241,201
9210 DATA 205,0,91,6,8,197,1,13,0,213,22
9,237,176,225,209,36,20,193,16,241,201
9220 DATA 62,2,205,0,91,235,60,205,18,91
,254,21,32,244,201
9230 DATA 62,21,205,0,91,235,61,205,18,9
1,254,2,32,244,201
```

Address	Hex Code	Assembler
7F80	012000	LD BC,0020h
7F83	110100	LD DE,0001h
7F86	6A	LD L,D
7F87	CDA97F	CALL 7FA9h
7F8A	3E3E	LD A,3Eh
7F8C	D7	RST 10h
7F8D	61	LD H,C
7F8E	10FE	DJNZ -2
7F90	25	DEC H
7F91	20FB	JR NZ,-5
7F93	6A	LD L,D
7F94	CDA97F	CALL 7FA9
7F97	3E20	LD A,20h
7F99	D7	RST 10h
7F9A	6B	LD L,E
7F9B	CDA97F	CALL 7FA9h
7F9E	3E3E	LD A,3Eh
7FA0	D7	RST 10h
7FA1	14	INC D
7FA2	1C	INC E
7FA3	7B	LD A,E
7FA4	FE20	CP 20h
7FA6	20E5	JR NZ,-27dec
7FA8	C9	RET
7FA9	3E16	LD A,16h
7FAB	D7	RST 10h
7FAC	AF	XOR A
7FAD	D7	RST 10h
7FAE	7D	LD A,L
7FAF	D7	RST 10h
7FB0	C9	RET

Table 8 Arrow Codes

Part 2

The TS 2000 Series

CHAPTER 7
Introducing the Timex

In the first section, I described the general concepts of microcomputers from first principles. Now it is time to examine the workings of the Timex 2000 computer specifically. There are two models available, but the only difference between them is that one contains 16K of RAM and the other 48K. The additional memory not only gives the scope for larger programs to be run and useful quantities of extra data to be stored 'on board', but also it allows the use of a further three types of video display. The hardware needed to generate these is contained in both machines, but bringing them into operation requires more memory space than the 16K computer can spare.

The Timex 2000 is a development of the Sinclair Spectrum, one of the first colour computers available at a realistic price to the average consumer in Britain. Much of its design was developed from its predecessors, the ZX80 and ZX81; notably, the 'single key entry' of BASIC. By retaining the same type of layout for their machine, Timex have been able to exploit the wide range of software that has emerged for the Spectrum in the UK, as it can be converted relatively easily to run on the 2000. Indeed, programs that consist exclusively of BASIC can often be loaded and run successfully on both computers. Machine code tends to demand a little more relocating in order to be compatible.

Features of the Timex include:

1. A high resolution video display that can be viewed on an ordinary domestic television set. It has a resolution of 256 by 192 dots, and it can print 32 characters in each of its 24 lines of text. Each character space can be displayed as two different colours, termed PAPER and INK: both of these can be one of eight colours: black, blue, red, magenta, green, cyan, yellow and white. In addition, any space can be set to BRIGHT, which increases the luminance; or FLASH, which swops PAPER and INK values at a fixed rate. Software allows drawing to an accuracy of one dot, or pixel, on the upper 22 lines of the screen, and the use of *user defined* graphic shapes. The screen data is *memory mapped,* meaning that the display is copied from RAM, which is addressed by the address bus; and it occupies over 6K of the memory that would otherwise be available to the programmer.

2. A BEEP facility which allows the creation of simple tones with BASIC, and more complex effects from machine code programs. The sound emerges from a small speaker built into the case. Although quiet, the ear socket can be used to feed an external amplifier.

3. An interface makes possible the use of an ordinary audio cassette player as a storage medium. BASIC programs, data and machine code can be saved on normal cassettes, and loaded back into the computer via the MIC and EAR sockets with the supplied lead. The cassette port uses a Schmitt Trigger in its circuitry, which enhances reliability so that data can be transferred at a rate of about 1200 *baud*; fast by comparison with some other machines. The creation of the serial data is handled by the system software, which also supports VERIFY, a command that checks that a good recording has been made.

4. A BASIC interpreter accompanies the machine's operating system in ROM: it is a development of the ZX80 and ZX81 BASIC with provision for multi-dimension arrays, a full expression evaluator, floating point arithmetic and syntax checking that occurs on line entry. String slicing is implemented by a TO function: for example, if A $ is 'Hello', then A $(2 TO 4) would be 'ell'. BASIC lines are entered from the keyboard by a *single key* method: each key has up to five different meanings for the operating system, depending on where in the BASIC line the key has been entered, and on the use of the shift keys. As a dialect of BASIC it is excellent for first time users.

5. A provision to drive a 'PORT MAPPED', low cost printer. The printer is controlled by software, thereby simplifying the interface. (Although of similar design, there are differences between the Timex and Sinclair printers which preclude the UK version from working on the Timex computer.)

6. Finally, a software provision is made to drive a 'string floppy' storage device. In Britain, this is a low cost item called a microdrive which uses high quality magnetic tape cartridges that travel at a fast speed. The performance approaches that of floppy disk drives, and the cost is less than half.

The rapid development of the home computer market meant that, by the time Timex decided to launch their own version of the Spectrum, it was necessary to improve the specification. The original design was developed to include the following features:

1. Two joystick connections that allow the use of any digital stick which terminates in a D type plug, and is wired in the same manner as the Atari type. A BASIC function (STICK) allows them to be read without resorting to machine code.

2. A sophisticated sound generating chip, in addition to the simple BEEP command. This allows sound to be played without the entire computer being dedicated solely to the sound while it lasts. The chip is the popular AY-3 8912, which can create three tone channels and a noise source, all with envelope control.

3. A bank switching arrangement that extends the addressing capacity of the memory map 'sideways'. Although the address bus is still only capable of generating 16-bit values, these can be applied to different banks of either RAM, ROM or additional devices.

4. A cartridge slot which accepts both *applications* (ie program) or *language* software: it is also able to host RAM. Cartridges of AROS (Application ROM Oriented Software) and LROS (Language ROM Oriented Software), are mapped into a different bank of the memory map, thereby freeing the system RAM for data storage. An LROS can make the computer function with a language different from the resident BASIC, without preventing access to the machine's operating system routines.

5. A provision to fit an expansion unit that allows additional RAM or other external devices to be memory mapped into one of the 253 available expansion banks. Software can recognise up to 13 distinct peripherals, including floppy and hard disks, networks, and 80-column printers.

6. Three additional modes of graphic display are available to 48K or 16K machines fitted with a suitably designed cartridge. These modes are as follows:

a. You can create a second display file, identical to the first and sitting just above it in memory. This doubles the overhead of RAM that the screen display consumes. Either of the displays can be sent to the screen, which allows for such techniques as animation and the completion of a picture before 'cutting' to it.

b. The use of both display files in conjunction, to create a High Resolution Colour display. This has the same pixel resolution as the standard screen, but the colour resolution is increased from 8 by 8 pixels to 8 by 1. Each character space is provided with eight attribute bytes which control the PAPER, INK, BRIGHT and FLASH values for every line of eight bits.

c. There is a 64-column mode which doubles the horizontal resolution to 512 pixels, allowing a line of text to contain 64 characters. This mode also uses both display file memory areas. You are restricted to a single PAPER colour for the whole screen, and whichever of the 8 you choose automatically sets a contrasting INK colour.

7. The Timex expands on the Spectrum's BASIC: SOUND allows the manipulation of the sound chip, and STICK reads the joystick ports. FREE is a function that passes on to the programmer whatever memory is still available; for example, DELETE 200,300 will remove all the program lines between the two specified numbers. RESET is not, as you may think, a hardware reset; it can specify which expansion bank is initialized, and perform a jump to that bank's start address. RESET* will cause the computer to act as if it had been turned off and on without losing the memory contents. RESET can also be used in combination with another command: ON ERR. This is capable of preventing the normal error reporting and can take a form such as ON ERR GOTO line number. ON ERR RESET cancels any prior ON ERR command. The string floppy commands of the Spectrum are fully implemented.

8. As for the exterior of the machine, the Timex is much improved: it has hard keys, a fullsize space bar and two CAPS SHIFT keys located either side of the layout as in a typewriter. A video socket provides an unmodulated feed for use with a monitor; the sound output and red, green and blue video signals (for running a high definition colour RGB monitor) are available in the edge connector at the back of the machine. There is also a power ON/OFF switch!

The above improvements to the already established and popular UK Spectrum computer make the Timex 2000 colour series a versatile and powerful design.

The structure of the hardware of the Timex is represented in **Figure 6**. If you were to compare this with the actual circuit board inside the case, there would not at first appear to be any resemblance: yet, apart from some small components that do cloud the picture, the main features can be identified. The most obvious is the microprocessor itself. This is a Z80 A: a version of the Z80 which is capable of running with a higher clock frequency than the standard model. An 8-bit wide data bus runs around the board from the microprocessor to the other main components, occasionally being buffered from conflicts by appropriate resistors and diodes. The 16-bit address bus does the same. The remaining major part of the system is a 68 pin package, about one inch square, which is mounted in a socket in the centre of the board: it bears a Timex part number, but it gives no clue as to its function. This SCLD

chip can be said to perform, with some exceptions, anything that the CPU is unable to do: it is a custom-manufactured workhorse that provides the means of generating a video display: it carries out all the port address decoding, the more exotic forms of address decoding; it provides the clock signal, and more. The initials represent Standard Cell Logic Design: this is a method of producing, from a standard product, custom-made integrated circuits for a specific job. The chip manufacturers make a package that contains all the building blocks required for a complex circuit: gates, flip-flops, invertors, buffers etc. The final task of interconnecting them, using a photographic etching method, is left until the required design is finalised with their customers. This makes the job of building a chip to fulfil a specific function much cheaper, especially if large numbers will be required.

There is nothing 'magical' inside the Timex SCLD: its role could be undertaken by a collection of logic chips, although the space these would occupy might demand a huge circuit board: the factors of size and cost work to the advantage of custom built chips. The SCLD is attached to both the data and address buses, and it also receives most of the Z80's control signals: with this information, it generates more specific signals of its own which control the whole computer. Found in close proximity to it on the board are a *crystal* and a small variable *trimmer*: these are the external components of the *oscillator* inside the SCLD, which together form the clock signal for the CPU and SCLD itself. Other connections to the 'workhorse' include the EAR input socket through which serial data can be fed in from a cassette recorder. Outputs include the video and colour signals; the feed to the MIC socket for recording data; and signals to activate the sound chip, extension memory banks and keyboard.

Two ROM chips are the next most obvious things on the circuit board. One of them is capable of storing 128K bits, that is, 16K bytes; the other, 8K bytes. Here is the home of the system software, permanently etched as connections in silicon. The chips are linked to both address and data buses; their CHIP ENABLE signals come from the SCLD.

A 48K machine possesses six RAM chips, the smaller computer only two. They are all identical, being the 4416 dynamic types or an equivalent. Two chips together provide 16K bytes: one stores the four low bits of any specified byte in that block of 16K addresses; the remaining four high bits are dealt with by the other chip. The first two RAMs of both Timex models are treated to their own address lines because the SCLD chips need to access that area of memory independent of the CPU, in order to collect the data to be sent to the screen. To facilitate this, chips known as multiplexers (74LS157) are used. They have two inputs for each bit of the address bus, one from the

Z80 bus and one fed directly from the SCLD: a control signal from the SCLD determines which of these addresses is passed onto the RAM chips in the sockets marked U6 and U7.

The sound chip, an AY-3 8912, connects to the data bus, and its sound output is routed to the speaker. As the chip is port addressed, access to its registers is via the SCLD. It is also responsible for interfacing the joysticks.

Feeding a domestic television requires a good deal of electronics. Not only must the colour and luminance signals be combined with suitable synchronizing pulses and coded into one *composite video* signal, but also that signal must be *modulated.* This process imitates the workings of a television transmitter; consequently the area of the circuit board which does the 'transmitting' is enclosed in a metal screening can, so that the computer will not appear on the airwaves. A weak but identical version of the signals picked up by your TV aerial is then passed down a screened lead to the aerial socket on your set. As you know, different TV stations broadcast on different channels: a small switch under the case allows the modulator to transmit on either channel 2 or 3.

Most of the remaining components are concerned either with buffering the various signals between the major areas of the circuit board and the connectors, or with providing the correct supply voltages without which the integrated circuits could not function. The voltage which is supplied by the external mains adaptor is 17½ volts, but many of the chips need only 5 volts to function, and any more would cause damage: the RAM chips require +5, +12 and −12 voltages, relative to their GRN pins.

A word of advice for the incurably inquisitive amongst you: if you take your machine apart to investigate inside, you will almost certainly invalidate any guarantee that the makers or suppliers offer, but you are unlikely to do any damage if you take care. Wait for the warranty to expire if you can possibly contain yourself. The computer is held together by seven cross-headed self-tapping screws located underneath the case: the three short ones come from the front edge. Having pulled out the power plug and removed the screws, turn the computer the right way up before lifting off the top half. Be gentle! The keyboard is connected to the lower part by a *very* fragile ribbon cable. This can be pulled gently from its socket to allow the complete removal of the top. Now you can snoop away to your heart's content; however, don't drop anything inside, particularly if it's metal. Three even shorter screws hold the printed circuit board to the lower part of the case, but there is little underneath other than copper tracks. When reinserting the tail of the ribbon cable before reassembling the computer, take great care not to kink it as it will bend all too easily and may fracture. And don't leave anything inside!

CHAPTER 8
The Memory Map

The most prevalent limitation that the majority of microprocessors impose on the design of computers is the size of the address bus. With 64K bytes at their disposal, early microcomputers had room to spare; but with the falling price of memory chips, and the demand for increased storage space, recent products often use all their available addresses. The Timex 2068, not only uses all the standard memory map, it has an extra 8K of ROM addresses which can be brought into play. The sophisticated methods used to provide bank switching, (the technique the 2000 series uses to enlarge its memory map,) are complex. For the sake of clarity, I will describe in stages, first the standard map, and then how it is extended.

When first switched on and working, the processor will communicate with the Home Bank of memory addresses. The allocation of this is as follows. Between addresses 0 and 16383 decimal (0000h to 3FFFh), a 16K ROM holds the majority of the system software, including the routines that provide input and output via keyboard, screen and speaker; and the BASIC language interpreter.

RAM is found between 16384 and the top of the map. In the 16K machine, this extends only as far as 32767 (7FFFh), and the remaining addresses are vacant. The 48K computer has RAM at all the addresses up to 65535 (FFFFh): on both machines, not all the RAM locations are available to the user. Study the Home Bank Memory Map in **Figure 7**, and you will see that, from 4000h upwards, a great deal of space is taken up before the location marked PROG, which is the storing place for the beginning of any BASIC program entered. (Please note that **Figure 7** is not drawn to scale.)

The pixel data for the screen is stored between 4000h and 57FFh, which is called Display File 1: this holds the information that remembers whether any dot of the screen is paper or ink. It makes use of a large area, 6K bytes, but that is the penalty for good graphics. Starting at 5800h is another file for display purposes: the *attributes file*. It is worth repeating that the normal colour resolution of the 2000 is low, and 300h bytes are sufficient for the data which instigates the SCLD chip to generate the colour part of the picture. Immediately following the attributes, is a 100h block of memory called the *printer buffer*: this is used in conjunction with software to drive the low cost printer available

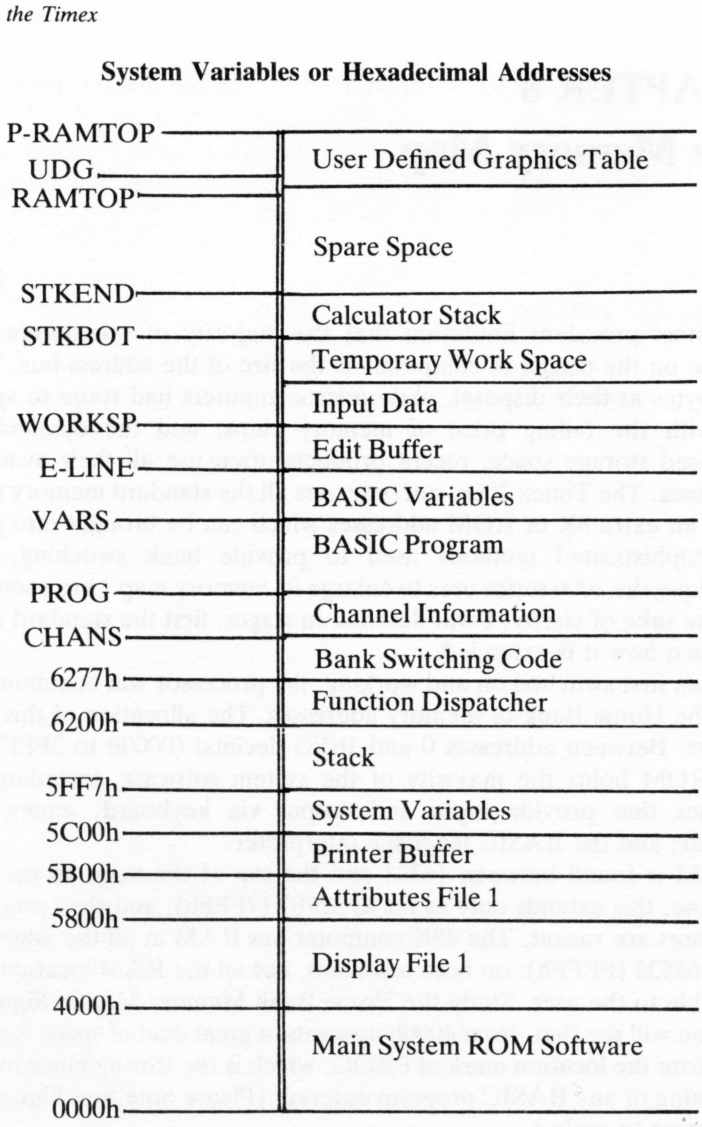


Figure 7 Standard Home Bank Memory Map

for Timex machines. It is in the buffer that a line of text is translated into eight lines of 256 pixels producing the character shapes. This data is then sent to the printer as a serial stream of ONs and OFFs, which control the stylus of the printer as it travels across the paper.

The next block of addresses, called the *system variables*, is concerned with the running of the machine's own system software. Any program

which is held totally in ROM is hampered by the fact that it cannot store variables within its own memory area, except for the few bytes it can keep in the registers of the processor, which are unused by the program itself. It may, of course, lodge the bytes on the stack, but retrieving them at random is impossible. Consequently, the area of memory from 5C00h to 5FFFh, is set aside for storing such information as the current colours to be used for printing to the screen; where the BASIC program has stored its variables; or even, where the BASIC program itself starts: (this moves about when extra equipment is attached to the computer.)

You will find a list of the system variables in the manual: this list indicates briefly what the data is that they store. Some can be altered by the programmer to achieve the desired effect: changing others will cause a peremptory crash of the whole machine. RAMTOP, the variable at 5C82h, for instance can be POKEd with a lower value to deceive the computer about the existence of memory locations. Normally, when the command NEW is carried out, all memory is wiped clean, but, with this device, the locations above your RAMTOP will not be affected. On the other hand, if you POKEd a different value into PROG, the address at which the BASIC program starts, the interpreter will fail to find the program when you RUN, causing a crash. (You may just be lucky and POKE in a value that is the start of a line in the program, but still, any earlier lines will be lost.)

It is worth noting that the values held by the two-byte system variables are stored in the manner of the Z80 itself: that is, the least significant byte first. Let me give an example that also shows the advantage of using hexadecimal numbers. PROG stores an address: it is a 16-bit number, so it needs two bytes of memory space, 5C52h and 5C53h. If the BASIC is in its normal place, then PEEKing these to values should provide a standard result. Try entering PRINT PEEK 23635, and PRINT PEEK 23636: you should get 86 and 104 respectively. In order to find out which addresses these represent, you need to multiply the second value by 256: PRINT 256 * PEEK 23636 will give you 26624. Now add the value in the first location by entering PRINT 256 * PEEK 23636 + PEEK 23635: this should elicit the result 26710, ie (256 * 104) + 86. If the Timex could be persuaded to give its answers in hexadecimal, (and programs are available offering this facility), the results of PEEKing the RAMTOP system variable would have been 56h and 68h. We can multiply a hex number by 256 through simply adding two zeros to the end: 68 becomes 6800h; add 56h, and we have 6856h. Guess the decimal version of 6856h! Merely by reading the two hex bytes in reverse, we arrive at the true two-byte value stored in RAMTOP. Admittedly, using hex numbers on a machine that cannot understand them is nonsense, but with the use of an appropriate Monitor program, many tortuous calculations can be avoided. If you

which is held totally in ROM is hampered by the fact that it cannot store variables within its own memory area, except for the few bytes it can keep in the registers of the processor, which are unused by the program itself. It may, of course, lodge the bytes on the stack, but retrieving them at random is impossible. Consequently, the area of memory from 5C00h to 5FFFh, is set aside for storing such information as the current colours to be used for printing to the screen; where the BASIC program has stored its variables; or even, where the BASIC program itself starts: (this moves about when extra equipment is attached to the computer.)

You will find a list of the system variables in the manual: this list indicates briefly what the data is that they store. Some can be altered by the programmer to achieve the desired effect: changing others will cause a peremptory crash of the whole machine. RAMTOP, the variable at 5C82h, for instance can be POKEd with a lower value to deceive the computer about the existence of memory locations. Normally, when the command NEW is carried out, all memory is wiped clean, but, with this device, the locations above your RAMTOP will not be affected. On the other hand, if you POKEd a different value into PROG, the address at which the BASIC program starts, the interpreter will fail to find the program when you RUN, causing a crash. (You may just be lucky and POKE in a value that is the start of a line in the program, but still, any earlier lines will be lost.)

It is worth noting that the values held by the two-byte system variables are stored in the manner of the Z80 itself: that is, the least significant byte first. Let me give an example that also shows the advantage of using hexadecimal numbers. PROG stores an address: it is a 16-bit number, so it needs two bytes of memory space, 5C52h and 5C53h. If the BASIC is in its normal place, then PEEKing these to values should provide a standard result. Try entering PRINT PEEK 23635, and PRINT PEEK 23636: you should get 86 and 104 respectively. In order to find out which addresses these represent, you need to multiply the second value by 256: PRINT 256 * PEEK 23636 will give you 26624. Now add the value in the first location by entering PRINT 256 * PEEK 23636 + PEEK 23635: this should elicit the result 26710, ie (256 * 104) + 86. If the Timex could be persuaded to give its answers in hexadecimal, (and programs are available offering this facility), the results of PEEKing the RAMTOP system variable would have been 56h and 68h. We can multiply a hex number by 256 through simply adding two zeros to the end: 68 becomes 6800h; add 56h, and we have 6856h. Guess the decimal version of 6856h! Merely by reading the two hex bytes in reverse, we arrive at the true two-byte value stored in RAMTOP. Admittedly, using hex numbers on a machine that cannot understand them is nonsense, but with the use of an appropriate Monitor program, many tortuous calculations can be avoided. If you

would like proof that 26710 is indeed the starting point of the BASIC program, then try the following procedure: enter a single line of BASIC into an otherwise vacant computer, perhaps, 100 REM the first and only line; then POKE the first two bytes of the program area with zeros, ie POKE 26710,0 and POKE 26711,0. Now list the program, and see the disruption caused: this is because each line is stored with the first two bytes, which represent the line number. If you POKE in other values, you will discover that, to add to potential confusion, these line numbers are stored the 'correct' way round: that is, the most significant byte first.

Returning to the memory map, let us examine what lies between the end of the system variables and the start of any BASIC program. A small machine code routine used in the setting up procedure can be found from 6000h to 6017h: its use, and the reason for its location here will be explained later. Next comes a crucial chunk of memory, used by the CPU as the stack. It 'grows' from the top down: early values passed to the stack by the Z80 will be stored near 61FCh; as more data is PUSHed onto the stack, the SP will point to lower addresses. There is plenty of room here; you can PUSH more than 250 two-byte values onto the stack before it begins to grow down into the system variables area. The stack is also used by BASIC to store Return addresses for Gosub returns.

Located next, at 6200h, is a machine code routine of interest to programmers who wish to use the Timex's resident system software for their own machine code programs. You need only PUSH the appropriate *service code* onto the stack and call this routine, the *utility function dispatcher*: it will call the required routine in the ROM for you. Unfortunately, this is not quite as simple to use as it may at first sound: I will go into more detail later. Further routines exist in the next block of RAM, concerned with the Timex's bank switching method of memory expansion. They take up nearly 1½K of memory: their purpose and operation is complex enough to necessitate a separate description.

Finally, before we reach the BASIC program area, there is a small table of data known as the channel information. The Timex uses a method to input and output data that assigns a channel number to each of its available types of communication. The device which is currently the active *stream*, is the one that receives and sends data. For example, if the screen is the active stream, then the data stored for that output channel in the channel information table, is used by an all purpose output routine to discover the whereabouts of the routines that deal with screen output. If you were to open the printer stream, the PRINT command would instigate the sending of what would normally go to the display files for transmission to the TV, to go instead to the printer buffer. This may seem an unnecessarily complicated way of working, but it comes into its own when extra channels are added to the system.

The bulk of the I/O (Input and Output) routines already exist, and by looking up the information for the current stream, they can be diverted as required to deal with each special case. The table is built up during the initialization process of the system, by checking which channels are present (including peripherals attached via an expansion bank), and making entries accordingly. Try the following program, which outputs to a stream other than the normal part of the screen. The # symbol, representing 'stream', is an extended and symbol shifted version of the 3 key.

```
10 PRINT # 1; 'Normally this would be at the top'
20 IF INKEY$="" THEN GOTO 20
```

The program illustrates what happens if you output data when stream one is selected: this is the lower part of the screen normally reserved for INPUT prompts and error messages. If you try other numbers, you will normally get an error message as there is no information for the output routine to find in the table regarding other streams.

We have now reached that part of the memory map which holds any BASIC program. The size of this area and the one immediately above, the variables area, will depend on the size of program loaded, and the number of variables it uses. Each time you enter a line of BASIC, the memory contents above the point that it is stored are shifted up to make room; and the relevant pointers that are held as system variables are altered accordingly. The next space is for editing lines of a program: when you select them, they are moved here and a copy sent to the lower part of the screen. As you move the cursor, you can make any changes, until ENTER is pressed: then the line with the matching number is found, deleted, and replaced by the new corrected line from the edit buffer. This is also where completely new lines are created as you type them, but before you press enter. Next up the map are spaces that can be expanded to hold temporary data, such as numbers entered in response to an INPUT prompt, or strings that are being manipulated. Before we reach the spare space, there is another type of stack: this is one used by the system software for handling numbers. As I mentioned earlier, the Timex employs a method to represent numbers that requires five bytes to store them: this allows it to manipulate values both small and large to a reasonable degree of accuracy. When handling these numbers, the computer uses the calculator stack.

We have now reached the free memory area, the starting point of which rises and falls as the areas below expand and contract during program entry, editing and run time. Finally, at the very top of the memory is a block of bytes: FF58h to FFFFh on the 2068; 7F58h to

7FFFh on its smaller brother. This holds the user definable graphic dot patterns which are at first filled with the shapes of their corresponding letters; but defining your own shapes will change their contents.

That is the general layout of the memory map of the Home Bank. It is worth noting that, at switch on, none of the RAM addresses hold anything relevant, so any data must be written in during system initialization. This task, and many others, are performed in the time between the moment of switching on with the appearance of the blank screen, and the printing of the copyright messages: not long in human terms, but for a computer, plenty of time for a great deal to be accomplished.

The next aspect of the memory map to appreciate is the way that the EXTENSION ROM comes to be addressed. It is interrelated with the DOCK bank, which is the memory map that applies to the cartridge socket. Take a look at **Figure 8**, and note how both the Home Bank and the DOCK bank are divided vertically into 8K chunks. It is possible to switch either vertical chunk into circulation so that, for instance, the first four chunks of the home bank and the last four of the DOCK bank are the areas of memory that are read and written from and to by the CPU. These chunks can be selected and deselected during program operation, so that a ROM plugged into the cartridge slot can be present in the memory, as seen by the Z80. You could also store data in the same chunk of addresses in the home bank, and, with the use of the bank switching routines, retrieve that data for use by the ROM program. The EXROM, or extension ROM, emerges when a routine sets bit 0 of port FFh. This operates a latching circuit that switches out of the DOCK memory map, the first chunk of memory plugged into the cartridge port (even if it's empty), and replaces it with the EXROM. Now, when chunk 0 of the DOCK bank is enabled, rather than the first chunk of the home bank, at these addresses the processor sees the contents, not of the ROM in the home bank, but those of the EXROM. What is in this EXROM, and why is it needed?

The system software of the UK Spectrum computer almost filled its available ROM addresses, and the embellishments provide for the Timex have meant that the space available in a 16K ROM is insufficient. A straightforward solution to this problem presents itself: increase the size of the ROM. Many of the latest home computers have more of their memory map tied up in this manner, sometimes as much as 32K, but this gives rise to the aggravation of less space for programs and data. If some form of bank switching is introduced, then it makes sense to reserve a large part of the home bank, the quickest and easiest to access, for RAM: consequently, the EXROM contains many of the routines that are not time-critical. If it held, say, the calculator routines, then each time a mathematical calculation were performed, time would be wasted

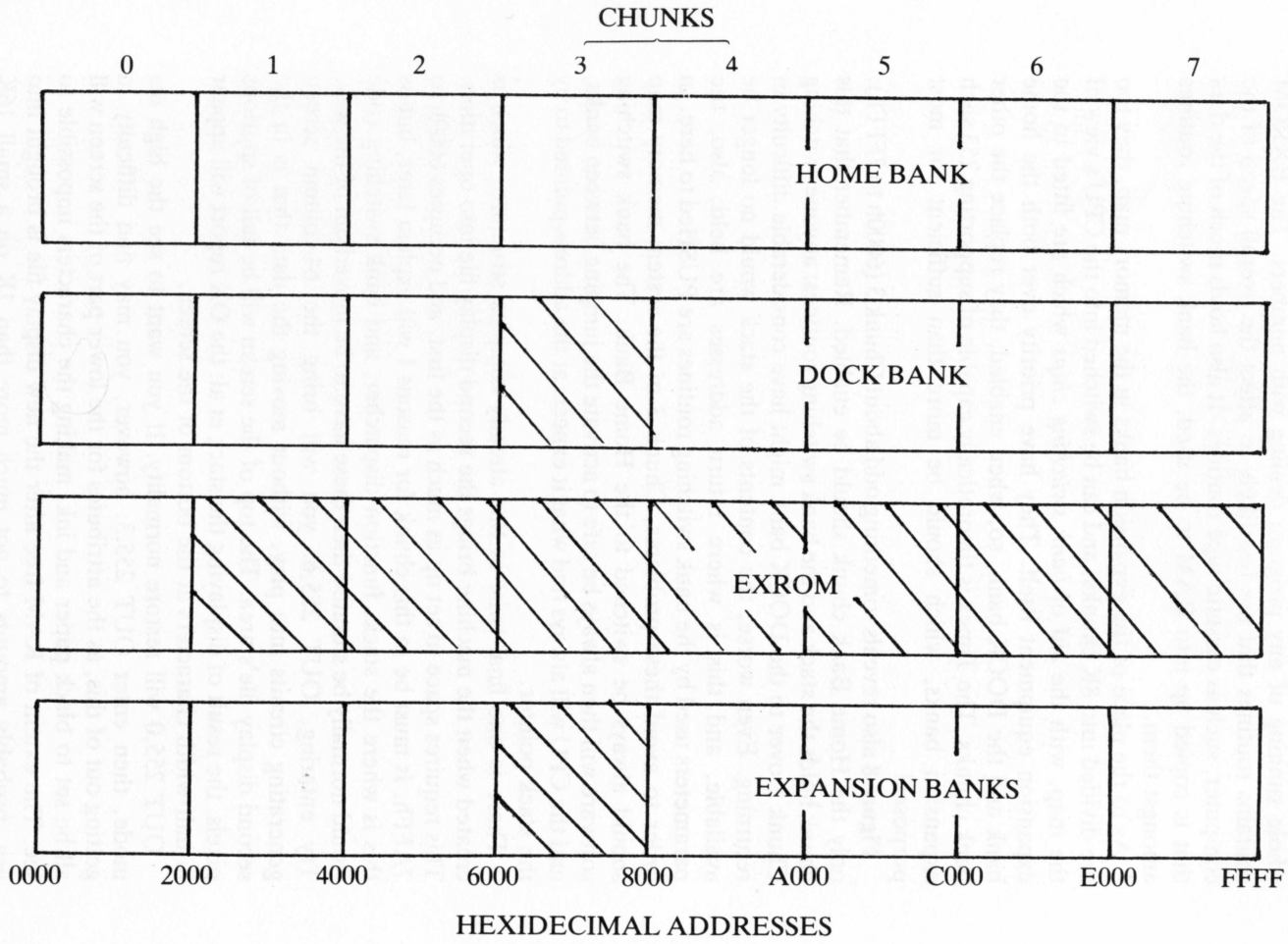


Figure 8 System Map

switching banks and transferring control, thereby slowing down the whole running of any program dealing with numbers. The EXROM contains routines that are less likely to affect the overall speed of the computer, such as cassette tape routines. It also holds much of the data that is copied up into RAM to be used, the bank switching routines amongst them.

As to the place of the expansion banks in the memory map, they too are divided into 8K chunks, and can be switched into the CPU's view of the map, with the aid of *bank switching chips* which are fitted to the expansion equipment itself. They have priority over both the home bank and the DOCK bank, so, when enabled, they replace the other bank chunks. The Timex is theoretically capable of supporting 253 such expansion banks, which should be more than sufficient for most purposes!

Figure 8 also reveals something odd about Chunk 3 (6000h to 7FFFh): only the Home Bank chunk should be enabled. Remember that this chunk holds the stack and the bank switching routines: anyone switching Chunk 3 over to the DOCK bank might have considerable difficulty in returning. Even worse, the contents of the stack would no longer be available, and this is where return addresses are held; also, the parameters used by the bank switching routines are PUSHed to here. In order to avoid these problems, Chunk 3 of the system memory map should always be switched to the Home Bank. The bank switching software will then always be safe to activate the jumping between banks, and the CPU will always find what it expects at the address pointed to by the stack pointer.

There is one final twist in this already complex structure, which is created when the machine brings the second display file into operation. This requires space to set up as much as the first, and occupies 6000h to 7AFFh. It must be in this chunk for reasons I will explain later, but as this is where the stack, function dispatcher, and bank switching code would normally be situated then these have to be moved out of the way. Try entering, OUT 255,6: you will bring the 64-column screen generating circuits into play, without moving the data that is in the second display file's area. The top of the screen will be full of spurious pixels, the result of displaying the stack et al: the OK report will appear in half-width characters at the bottom of the screen.

OUT 255,0 will restore normality. If you want to see the high res mode, then enter OUT 255,3: however, you may find difficulty in getting out of this, as the attributes for the lower part of the screen will all be set to black paper and ink, making the characters impossible to see. The extent of RAM free after the new display file is brought into use, probably amounts to not much more than 1K on a small 16K Timex, so a very sophisticated cartridge will be required if the extra

display modes are to be used. The 48K machine still has adequate room in RAM, so a routine exists in the EXROM to move the stack, the bank switching routines, and the function dispatcher up to the top of memory; and to change the system variables, and stack pointer accordingly. A system variable called VIDMOD, records when this has been accomplished, and the operating system checks this before attempting to use the bank switching routines.

We have come to the end of our guided tour round the memory map. Much of the above will be expanded upon soon, but now it is time to look at how the computer achieves its communication with the outside world.

CHAPTER 9
The Keyboard

A computer can be of little use unless it has some means of receiving information. When a Timex 2000 series computer first emerges from its packaging, there are four ways of feeding information to it: cartridge, cassette, joystick or keyboard. You can plug into the slot on the right a cartridge containing a complete program or load the memory with data from a cassette player. However, the programs would almost certainly need some other data. This could come from the joystick ports but the most likely source would be the keyboard. Even when the screen flashes with the familiar message, 'Press any key to begin', it is expecting data in its most simple form, from a key.

The way keyboards actually feed information into a microcomputer varies with each machine. Some keyboards can interrupt whatever the CPU is doing, and call attention to themselves whenever they have something to pass on: what then happens depends on the software that has been built into the machine.

The Timex uses a slightly different approach. The keyboard itself is a completely passive device, but the operating system scans it at very short, regular intervals to ascertain whether any key is being pressed: if this is the case, it stores the value in a particular memory location, which the system's software can read and respond to as required. In order to understand how these regular intervals occur, we shall first need to look at some of the functions of the Z80 processor which I have so far failed to mention.

The two pins that concern us are called NMI and INT: they empower a microcomputer designer to achieve a machine which, although it is still able to do only one thing at a time, can nevertheless share that time between different tasks. For example, some printers, such as the old Teletype terminals which you may have seen (and heard!), are very slow. When they are sent a character to print there is a considerable delay, in computer terms, before the next one can be sent. Instead of having your microcomputer standing idle during this time, you can employ it elsewhere; whilst the printer can let it know that it is ready for another character by putting a signal on the INT pin.

So what's all this nonsense about ancient printers and impatient Z80s? While the interrupt system is available, it can be used to make your

CHAPTER 9
The Keyboard

A computer can be of little use unless it has some means of receiving information. When a Timex 2000 series computer first emerges from its packaging, there are four ways of feeding information to it: cartridge, cassette, joystick or keyboard. You can plug into the slot on the right a cartridge containing a complete program or load the memory with data from a cassette player. However, the programs would almost certainly need some other data. This could come from the joystick ports but the most likely source would be the keyboard. Even when the screen flashes with the familiar message, 'Press any key to begin', it is expecting data in its most simple form, from a key.

The way keyboards actually feed information into a microcomputer varies with each machine. Some keyboards can interrupt whatever the CPU is doing, and call attention to themselves whenever they have something to pass on: what then happens depends on the software that has been built into the machine.

The Timex uses a slightly different approach. The keyboard itself is a completely passive device, but the operating system scans it at very short, regular intervals to ascertain whether any key is being pressed: if this is the case, it stores the value in a particular memory location, which the system's software can read and respond to as required. In order to understand how these regular intervals occur, we shall first need to look at some of the functions of the Z80 processor which I have so far failed to mention.

The two pins that concern us are called NMI and INT: they empower a microcomputer designer to achieve a machine which, although it is still able to do only one thing at a time, can nevertheless share that time between different tasks. For example, some printers, such as the old Teletype terminals which you may have seen (and heard!), are very slow. When they are sent a character to print there is a considerable delay, in computer terms, before the next one can be sent. Instead of having your microcomputer standing idle during this time, you can employ it elsewhere; whilst the printer can let it know that it is ready for another character by putting a signal on the INT pin.

So what's all this nonsense about ancient printers and impatient Z80s? While the interrupt system is available, it can be used to make your

Timex appear schizophrenic. Every sixtieth of a second, the ULA chip sends out a video signal, starting with a 'frame pulse' to which the TV or monitor synchronises. Suffice it to say that, in the process of generating video, the ULA creates a pulse of zero volts which it sends to the CPU's INT pin every sixtieth of a second. The Timex software has configured the Z80 to respond to this signal in the simplest manner available. When the Z80 has finished performing each instruction, it checks its interrupt input; if it senses a negative pulse, it saves the contents of the program counter (on the stack), and loads it with 0038h (56 decimal).

We have now directed the processor from whatever it was doing and pointed it towards the machine code routine stored at 0038h in the ROM. This routine does three things. Firstly, it increments the system variable called FRAMES: this is a three-byte value held in RAM locations 5C78h to 5C7Ah, which is used to measure time through the BASIC command 'Pause'. Secondly, the routine scans the keyboard to see whether any key is depressed; in which case, it decodes the key, taking into account shift keys and, unless two or more keys are pressed, it stores the code in RAM location 5C58h. Finally, the old value of the program counter is retrieved from the stack and reloaded; thus the Z80 returns to exactly where it left off before the interrupt occurred. The program ensures that the CPU registers are not changed: any that are used, have their previous values stored and then reinstated once the interrupt has finished, in order that the program that has been interrupted is not at all affected. The operating system can take action on the keypress if it wishes: it may be in command mode and react by interpreting the key as an instruction; or it may be running a program and therefore ignore the key completely.

When you can follow machine code, the main scanning routine will be of interest. It can handle ROLLOVER, which is easier to demonstrate than to explain: switch on your machine and clear the copyright message by pressing PRINT. Now hold down the P key, which will autorepeat and print a stream of P's to the screen. Now press another key as well. The repeat printing stops because the machine does not know which key you wish it to send to the screen. Lift your finger off the P key and the second letter will appear. The automatic repeating is also handled by the scanning routine.

Now to explain how the keys are connected. There are 41 keys and a full size space bar, but the CAPS SHIFT key appears twice, and the BREAK key is connected in parallel with the space bar, so in effect, there are 40 switches. These form a matrix connecting the 8 high address lines with 5 lines numbered KBD 9 to KBD 13. Circuits within the SCLD sense when the Z80 is performing an IN (FEh) operation, and respond by placing the voltages from the KBD lines onto the low five bits of the data bus. A study of **Figure 9** will show how this is achieved;

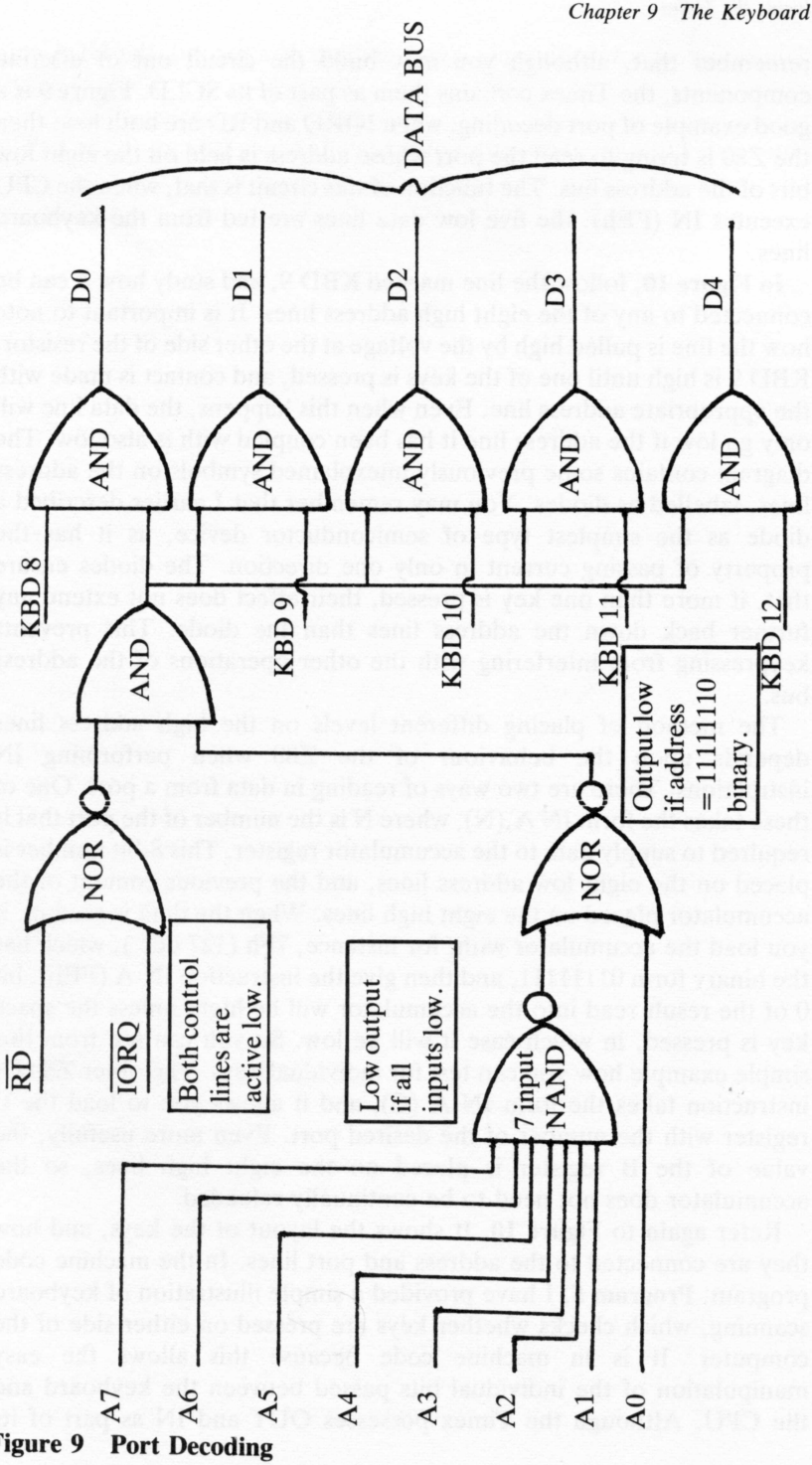


Figure 9 Port Decoding

remember that, although you may build the circuit out of discrete components, the Timex contains them as part of its SCLD. **Figure 9** is a good example of port decoding; when IORQ and RD are both low, then the Z80 is trying to read the port whose address is held on the eight low bits of the address bus. The function of this circuit is that, when the CPU executes IN (FEh), the five low data lines are fed from the keyboard lines.

In **Figure 10**, follow the line marked KBD 9, and study how it can be connected to any of the eight high address lines. It is important to note how the line is pulled high by the voltage at the other side of the resistor: KBD 9 is high until one of the keys is pressed, and contact is made with the appropriate address line. Even when this happens, the data line will only go low if the address line it has been coupled with is also low. The diagram contains some previously unexplained symbols on the address lines, labelled as diodes. You may remember that I earlier described a diode as the simplest type of semiconductor device, as it has the property of passing current in only one direction. The diodes ensure that, if more than one key is pressed, their effect does not extend any further back down the address lines than the diode. This prevents keypressing from interfering with the other operations of the address bus.

The method of placing different levels on the high address lines depends upon the behaviour of the Z80 when performing IN instructions. There are two ways of reading in data from a port. One of these takes the form IN A,(N), where N is the number of the port that is required to supply data to the accumulator register. This 8-bit number is placed on the eight low address lines, and the previous content of the accumulator placed on the eight high lines. When the data is read in, if you load the accumulator with, for instance, 7Fh (127 dec.), which has the binary form 01111111, and then give the instruction IN A (FEh), bit 0 of the result read into the accumulator will be high; unless the space key is pressed, in which case it will be low. So you can see from this simple example how you can test for individual keys. The other Z80 IN instruction takes the form IN A (C), and it allows you to load the C register with the number of the desired port. Even more usefully, the value of the B register is placed on the eight high lines, so the accumulator does not need to be continually reloaded.

Refer again to **Figure 10**. It shows the layout of the keys, and how they are connected to the address and port lines. In the machine code program, **Program 5**, I have provided a simple illustration of keyboard scanning, which checks whether keys are pressed on either side of the computer. It is in machine code because this allows the easy manipulation of the individual bits passed between the keyboard and the CPU. Although the Timex possesses OUT and IN as part of its

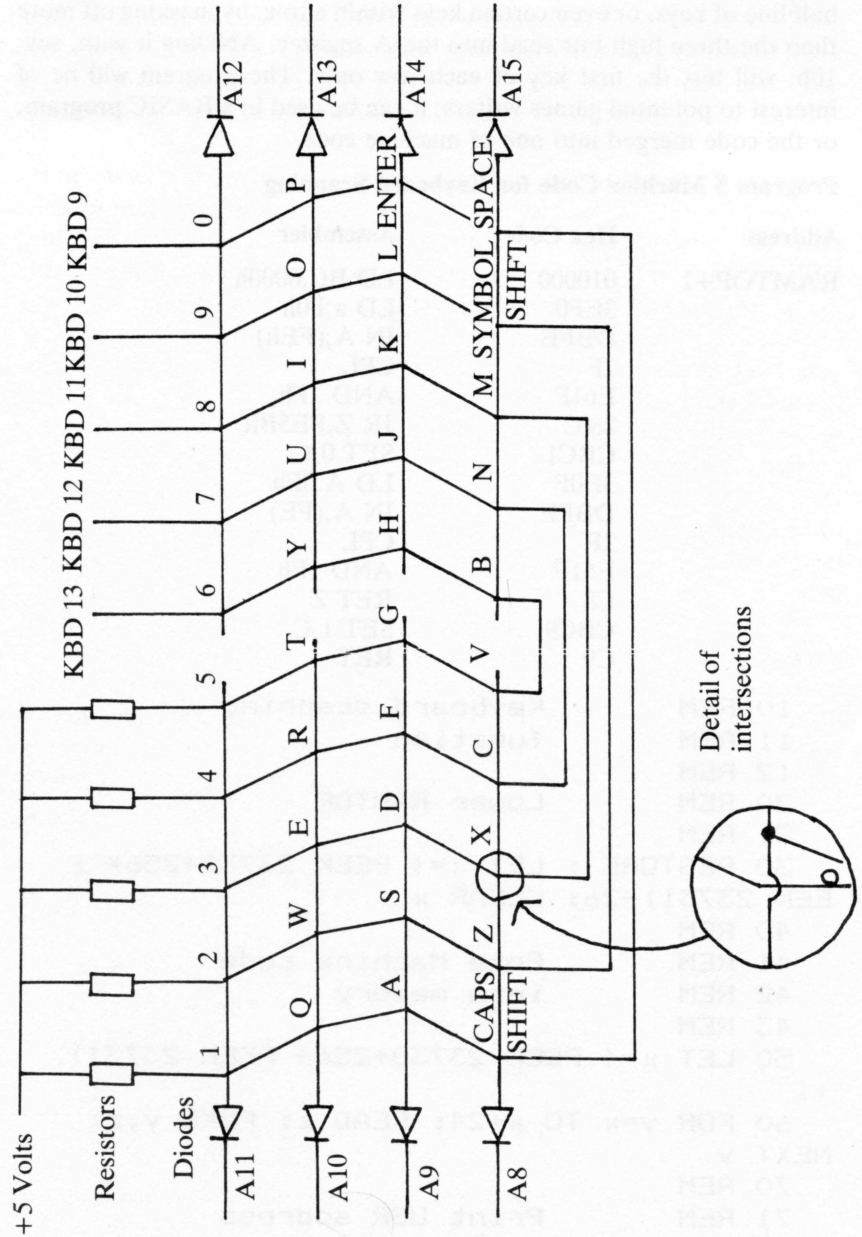

+5 Volts
Resistors
Diodes
A11
A10
A9
A8
A12
A13
A14
A15
KBD 13 KBD 12 KBD 11 KBD 10 KBD 9
1
2
3
4
5
6
7
8
9
0
Q
W
E
R
T
A
S
D
F
G
CAPS SHIFT
Z
X
C
V
Y
U
I
O
P
H
J
K
L ENTER
B
N
M SYMBOL SHIFT
SPACE
Detail of intersections

Figure 10

BASIC, testing individual bits can be slow and awkward as their arguments are in decimal. You could modify **Program 5** to test for any half line of keys, or even certain keys within a row, by masking off more than the three high bits read into the A register: ANDing it with, say, 10h, will test the first key of each row only. The program will be of interest to potential games writers; it can be used in a BASIC program, or the code merged into one of machine code.

Program 5 Machine Code for Keyboard Scanning

Address	Hex Code	Assembler
RAMTOP+1	010000	LD BC,0000h
	3EF0	LD a,F0h
	DBFE	IN A,(FEh)
	2F	CPL
	E61F	AND 1Fh
	2802	JR Z,FE5Bh
	CBC1	SET 0,C
	3E0F	LD A,0Fh
	DBFE	IN A,(FE)
	2F	CPL
	E61F	AND 1Fh
	C8	RET Z
	CBC9	SET 1,C
	C9	RET

```
  10 REM        Keyboard scanning
  11 REM        function
  12 REM
  20 REM        Lower RAMTOP
  21 REM
  30 RESTORE : LET x=( PEEK 23730+256* P
EEK 23731)-26: CLEAR x
  40 REM
  41 REM        Poke Machine code
  42 REM        into memory
  43 REM
  50 LET x=( PEEK 23730+256* PEEK 23731)
+1
  60 FOR y=x TO x+24: READ z: POKE y,z:
NEXT y
  70 REM
  71 REM        Print USR address
  72 REM
  80 PRINT  AT 1,2;"The routine is calle
d by the"; TAB 7;"function USR ";x
```

```
 90 REM
 91 REM        Example
 92 REM
100 PRINT  AT 8,10;"Press any keys"
110 PRINT  AT 12,14;: LET a= USR x
120 IF a=0 THEN  PRINT "NONE "
130 IF a=1 THEN  PRINT "LEFT "
140 IF a=2 THEN  PRINT "RIGHT"
150 IF a=3 THEN  PRINT "BOTH "
160 PRINT  AT 16,10;"USR ";x;" = ";a
170 GO TO 110
180 REM
181 REM        Machine code data
182 REM
190 DATA 1,0,0,62,240,219,254,47
200 DATA 230,31,40,2,203,193,62,15
210 DATA 219,254,47,230,31,200,203,201,
201
```

Note that, in its simplicity, it has one 'bug': the CAPS SHIFT key on the right is registered as if it were on the left, as it is connected in parallel with the other caps shift key. Also notice that the space bar appears on the right as it is in parallel with the BREAK key. Incidentally, the three high bits that are read in along with the five keyboard bits should always be masked off. On early models of the Spectrum, they always returned zeros, and many professional programs simply ignored them. When the second and third issue machines appeared, much embarrassment ensued, as these bits took on a random nature causing spaceships, monsters and the like to roam about on their own accord!

What if you want to use the built-in scanning routine for your own purposes? Remember that it is interrupt driven: the maskable interrupt can be disabled, in machine code, with the aid of the DI instruction, which causes the CPU to ignore its MI pin. Or the signal from the SCLD can be switched off: this is achieved by setting bit 6 of port FFh, and the SCLD will take care of the rest. Enter OUT 255,64. I'm afraid you will need to reset the machine afterwards. There are a number of ways to elicit a value from the keyboard; the choice of method depends on what kind of program you are writing. If you leave the interrupts enabled, you effectively open the door to the tamperings of machine code buffs: it may also upset any critical timing involved. But if you do decide to leave them enabled, then proceed as follows: test bit 5 of FLAGS, the system variable stored at 5C3Bh (23611 dec). If it is set, then a key has been pressed since the last time that flag was reset. You will find the

ASCII code for that key in the system variable location, LAST K, 5C08h. If you are only interested in the caps shift value, it is stored in location 5C04h (23556 dec), and it is updated every interrupt, so you will not need to test Flags. On the subject of CAPS SHIFT, you can set CAPS lock from within a BASIC or machine code program, by setting bit 3 of FLAGS 2 (5C69h). To see this happen, enter POKE 23658,8: INPUT A$.

It is possible that an interrupt will occur between testing FLAGS and fetching the value which may produce the occasional spurious result. In a machine code routine, this can be circumvented in two ways. You can ensure that the CPU marks time with the HALT instruction: it will perform NOP until it receives an interrupt. By inserting a HALT in the code immediately prior to reading the system variables, you gain a breathing space of one sixtieth of a second before the next interrupt. The alternative method is to disable the interrupts altogether and call the scanning routine yourself. As it is located at 38h, the RST 38 OP code will do the job in one byte. The system variables will be set accordingly, and the A register will hold the value from LAST K. The scanning routine contains an EI instruction, so you may need to use DI again.

It is worth noting that there is another keyboard scanning routine contained in the ROM, which tests the keyboard to see if BREAK is pressed. When it runs a BASIC program, or performs some commands, such as SAVE or LOAD, the Timex makes frequent calls to this routine in order to allow the user to stop the machine and return to the command mode. If you want to use the BREAK testing routine, it can be found with the aid of the function dispatcher. It returns the CARRY flag reset to zero if both BREAK and CAPS SHIFT are pressed. This subroutine is quite short; you may wish to copy it for your own purposes, and perhaps modify it to respond to other keys.

The keyboard is not only capable of producing ASCII codes: each key can also generate the BASIC tokens that represent commands and functions. This is dependent upon what mode the keyboard is in, as shown by the cursor: when it is a flashing K, then the next keypress is translated by the operating system as a command. Pressing P in the command mode generates the value F5h. It only appears on the screen as PRINT because the output routine recognizes it as a special case, and it looks up in a table (stored in the ROM from 98h to 226h), what string of characters to send to the screen, or for that matter, what is the currently selected output stream. So the Timex saves space and time: it does not need to store five bytes when it represents PRINT, and need only decipher one byte for each command or function. And we need never concern ourselves with the spelling of words such as RANDOMIZE!

You may expect me to cover how the joysticks input information to the computer, at this juncture; and logically, I should. However, their use is inextricably tied up with the sound chip, and they will be much easier to explain after that chip's functions have been unravelled.

CHAPTER 10
Pictures on the Screen

If a certain Scottish electronics' pioneer had got his way, the television set sitting in the corner of your room would contain a disc with lenses around its edges, spinning at great speed. Fortunately for the video industry, history has left us with a much better system. Both Baird's spinning discs and today's electronic replacement have the same principles of *scanning* at their roots: the only disadvantage of the modern method is that it is harder to understand.

The tube which displays the picture we see on the television set is called a Cathode Ray Tube. It is represented in cross section by **Figure 11**. The air is evacuated from inside the tube and the front face coated with small particles called *phosphors*. At the back of the tube is an *electron gun*: here a piece of metal called a *cathode* is heated up, and a high enough voltage is generated between it and the phosphors to cause the electrons at the cathode, already excited by the heat, to travel to the highly attractive positive voltage area that exists at the phosphors. As there is nothing to impede their path, a stream of electrons will flow through the vacuum to their goal. A magnetic field, and other voltages, can have a marked influence on the path taken by these electrons, so it is possible to use electromagnetic coils, and extra cathodes and *anodes* (the positive version of cathodes), in order to deflect and focus the *beam* of electrons travelling to the phosphors. This beam can be made to arrive at any particular spot on the screen. The special properties of the phosphors are significant: if bombarded with lots of electrons, a phosphor will give a glow of light. Within certain limits, the more electrons the greater the glow given off — very useful indeed!

So we have a method of lighting up any particular point on the TV screen with the application of suitable control voltages to the various components of the cathode ray tube. It is now a relatively small step to perceive how we might 'paint' a picture by aiming the beam at various parts of the screen and making them glow, moving on to others and returning before the original glow has subsided. **Figure 12** shows how this is done. The screen area is divided up into horizontal lines and a line is scanned. The intensity of the beam varies, resulting in the different brightnesses required. The beam is then reduced in intensity as it 'flies back' to the start of the next line and the process is repeated. When the

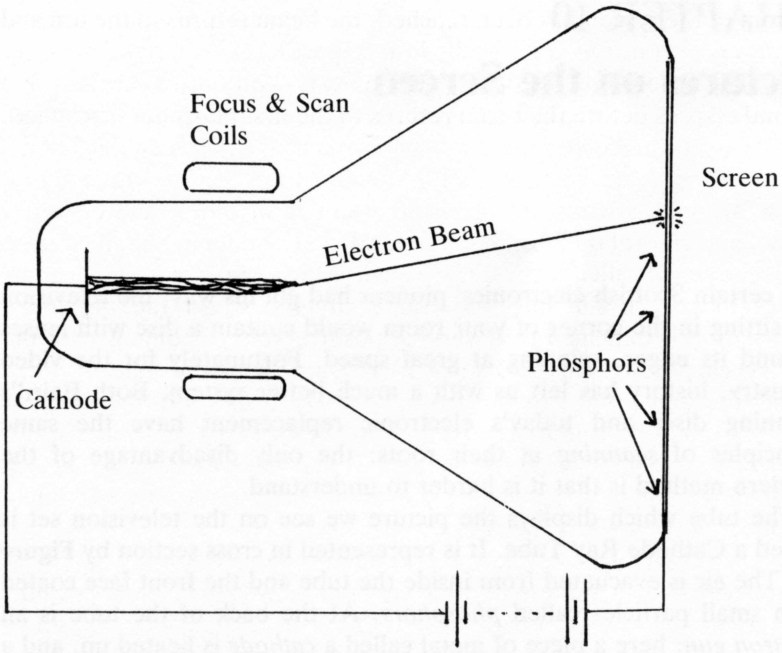


Figure 11 TV Tube

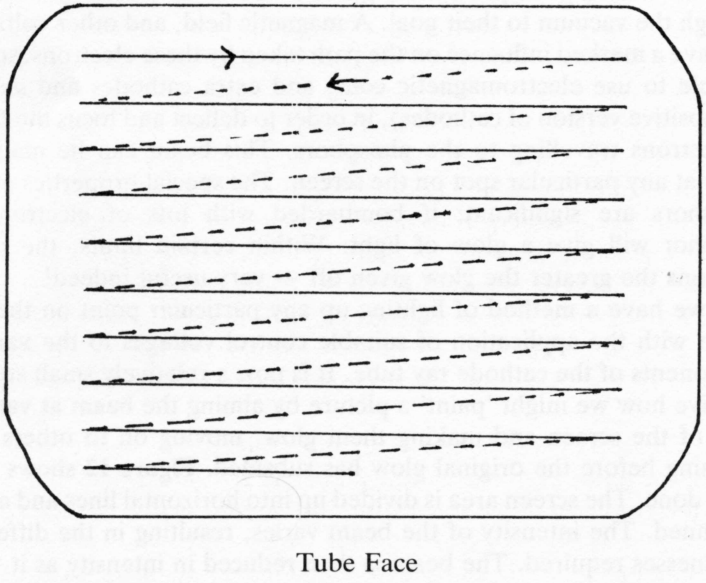


Figure 12 Raster

bottom of the screen has been reached, the beam returns to the top and begins again; but this time it fills in the spaces between the first set of lines. This whole process occurs very quickly, and only a thirtieth of a second elapses before the beam returns to the first phosphor it scanned.

In order to make an intelligible picture out of the screen display, its associated circuits must be sent a number of pieces of information. These are pulses which instigate the scanning of a new line or a new *frame*, and control the brightness generated. The data is contained in an analogue signal called the VIDEO signal, which employs negative going pulses to govern the line and frame circuits: whilst the brightness information is managed by a varying positive voltage coming after the line synchronizing pulses. **Figure 13** is a graph of one line of video signal: the signal that emerges from the VIDEO socket of the Timex is in this form. The signals received over the airwaves or from the TV socket of the computer are video signals that have been modulated in the same way as radio signals, so that they can be transmitted and then decoded by the *tuner* circuits in the television set.

As for colour, it makes matters even more complex. Any colour can be made up from a mixture of three very pure sources of primary colour: red, green and blue. A colour television has three kinds of phosphors coated on the screen: the tube contains not one but three electron guns, and masking between the cathodes and the screen ensures that the beam from one gun only lands on one colour of phosphor. The size of the different colour areas is so small that from a distance they merge into one source of light, to the human eye. With all three guns set to maximum, the screen will still glow white, but if the relationship of the output of the guns is varied, different colours are displayed. Special colour monitors will only require the three colour signals to be sent separately and they will achieve a very high quality of display. Normal televisions expect the colour information to be included with the video signal in what is known as a *coded composite video* signal, as this is how the broadcasting system transmits colour. It means that the channel takes up no more airspace and monochrome receivers can share the same signal. The system used to code the colour information is complex: suffice it to say that colour *difference* signals are modulated and mixed with the video in much the same way that more than one TV channel can transmit on the airwaves, and a tuner circuit can distinguish between them.

The above sparse description should at least familiarize you with some of the terms associated with video, and give you a clear idea of the way a television set draws its picture, by scanning across the screen at a very fast rate. Let's now return to your Timex and see how it goes about generating the video signal to feed a monitor, or via its modulator to a television set.

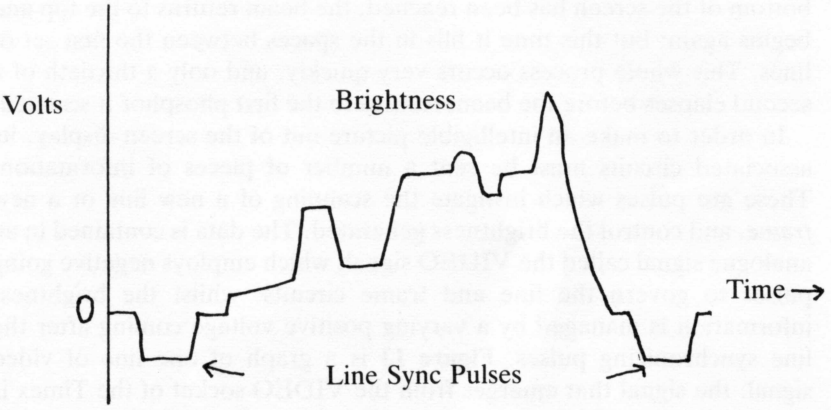


Figure 13 Video Signal

I mentioned earlier that much of the contents of the SCLD are concerned with the screen display: the data for the generation of that display is stored in various files in the area 4000h to 7AFFh. Output from the custom chip includes the SYNC pulses, and the analogue colour and video levels. Timing for the pulses is provided by dividing down the clock frequency. The CPU can signal to the SCLD information that controls some of the aspects of how the video is generated. By writing to the three low bits of port FEh, the colour used for the border can be changed: the value corresponds to the colour numbers used by BASIC: 0 is black, 1 is blue, and so on up to 7, which is the equivalent of white. This exemplifies the colour mixing: the three bits each control a primary colour, so if we mix red and green by sending 110 binary to the port, the colour displayed for the border is yellow, the same result as a mixture of red and green light. Writing to port FFh controls the type of display generated: bit 0 determines which display file is used, and setting it brings the second file into use. Setting bit 1 will cause the SCLD to use both files to produce the high resolution display; setting both bits 1 & 2 will evoke the 64-column mode, in which the paper colour is controlled by bits 3, 4 & 5. The convention used for border colour also applies to the paper colour, and the ink colour chosen by the SCLD is the binary complement. For example, to set a red paper we write 010 to bits 3–5, and the complement of 010 is 101: this is the value 5, so the SCLD chip will use red paper and cyan ink attributes.

Let's assume that the Timex is in the normal video mode: what is going on in the SCLD? Completely independent of the processor, it produces a frame pulse to commence the picture at the top of the television screen, and then it sends out line pulses at the right intervals. For the analogue video information, it uses the border bits as the data to

generate the corresponding colour and brightness levels, for insertion between the line pulses of the composite video. After sending out a number of lines containing the border, the SCLD starts the first line of the screen proper. A line synchronizing pulse is followed by a bit of border, and then the action starts: switching over control of the low 16k of RAM address lines to itself via the multiplexing circuit mentioned, the SCLD reads in the first byte stored in the display file, and also the first byte in the attributes file. Then it returns control of chunks 2 and 3 of RAM to the CPU. A problem that could arise here is the CPU also trying to read or write to the same block of memory when it cannot control the address lines: it is impossible for it to fetch any meaningful data if it attempts to read from that block simultaneously with the SCLD. To prevent this, the address bus is monitored by the SCLD, and if it senses that a clash is about to occur, it simply freezes the CPU by stopping the clock pulse which it is sending, until it has relinquished control of the address lines.

The information relating to the first eight pixels of the screen is now in the SCLD chip, each bit of the byte from the display file representing 1 for ink or 0 for paper. The attributes file controls the colour information as follows: bits 0, 1 and 2 determine the colour of the ink pixels; bits 3, 4 and 5, the paper. If bit 6 is set this signifies BRIGHT and the whole of the eight pixels are 'sat up'; that is, the luminance of both colours is increased. Bit 7 is the flash control: if set to 0 then the pixels are unaffected; if 1, then the output of a low frequency oscillator circuit in the SCLD determines if the paper and ink values are to be swopped over before coding.

The first eight pixels have been conveyed to the screen, and the process is repeated another 31 times until the entire top row has been sent then the border colour is reinstated for the remainder of the line. The SCLD proceeds to reiterate this process for the rest of the screen: however, the next line of pixels is not stored immediately after the first. I will show how the memory map of the screen is arranged in a moment. Each line continues to be fetched and transmitted until the bottom of the screen is reached, when more lines of border are produced, until it is time for the next frame pulse: and off we go again. So the SCLD happily transfers the contents of the display files to the outside world: red, green and blue colour signals are sent to the rear edge connector; composite video is available at the monitor socket; and a broadcasting lookalike is sent to the television set. When its access to memory in the region 4000h to 7FFh would upset the work of the CPU, then it stops the supply of clock pulses until the danger is over.

The other display modes differ in the way the files are read. If the second set of files is enabled, then data is fetched from addresses 2000h higher up: instead of starting at 4000h, the first byte is fetched from

6000h. In the high res mode, a new set of attributes is used for each byte of pixels: this requires a much larger attributes file which is stored in D-FILE 2. While operating in the 64-column mode, pixel data is stored in both display files. Although the SCLD has to put this out at double speed in order to provide the horizontal resolution, no attributes are needed because these are set by the SCLD.

Now to move on to the actual arrangement of the standard display files. At first glance, the method used seems remarkably convoluted. However, it is important that the SCLD should be able to scan this area with the minimum of decoding work. It is, therefore, understandable that the pattern used makes more sense when we study the addresses in binary. The screen is divided into three sections, each of which uses 2K of memory to store the pixel data. These sections occupy 4000h to 47FFh for the top of the screen; 4800h to 4FFFh for the middle, and 5000h to 57FFh for the bottom. Each section contains eight rows of characters. The top line of the first of these rows of characters is held by the first 32 bytes of each section. The next seven blocks of 32 bytes each, hold the same data for the remaining seven text rows of the section. Subsequently, the lines of pixels for the rows of characters are stored in the same manner: that is, all the second lines, then all the third lines, and so on, for all the eight lines of pixels that make up a row of character spaces.

This is complicated to describe in words and so it is best illustrated by a simple program. Enter **Program 6** and run it. It will ask for two values

Program 6 Screen Layout

```
  10 REM       Screen layout
  11 REM       examination
  12 REM
  20 REM        Set up
  21 REM
  30 INK 0: PAPER 7: BORDER 7: FLASH 0:
BRIGHT 0: CLS
  40 REM
  41 REM       Get values
  42 REM
  50 INPUT "Enter byte (0>255) ";byte
  60 IF byte<0 OR byte>255 THEN  GO TO 5
0
  70 INPUT "Enter color (0>255) ";color
  80 IF color<0 OR color>255 THEN  GO TO
 70
  90 REM
```

```
  91 REM       Alter Screen
  92 REM
 100 FOR x=16384 TO 22527: POKE x,byte:
NEXT x
 110 REM
 111 REM       Alter Colours
 112 REM
 120 FOR x=22528 TO 23295: POKE x,color:
 NEXT x
 130 REM
 131 REM       Display Values
 132 REM
 140 PRINT  AT 2,4;"Byte = ";byte; AT 6,
4;"Color = ";color; AT 10,4;"Press a key
 to alter"
 150 REM
 151 REM       Wait for keypress
 152 REM
 160 IF  INKEY$ ="" THEN  GO TO 160
 170 GO TO 10
```

to use for its demonstration: I suggest that you try 255 as the first one; and any 8-bit number for the second. It will then fill the entire display file sequentially with the first byte; and the three sections of the screen will in turn be filled with ink bits. Notice the 'venetian blind' effect as the character rows are gradually built up. When the screen is full, something slightly more dramatic will occur: the program POKEs the second value into the attributes file which sits immediately above D-FILE 1. This is effected much more quickly due to the smaller size of the file. I have said that things can sometimes be easier in binary, and here is a good example of this. If we write a 16-bit number to represent the screen locations, and use symbols to denote which bits tally with which parameters, it looks like this:

010SSLLLRRRBBBBB

The letters represent the following: S is the section number with 0 at the top and 2 (10 binary) at the bottom; L is the number of the line of pixels within the row of characters (a 3-bit number can stand for one of 8 lines); R is the row number; and B indicates the position of the byte in the row, the number of the column. The address splits neatly into two bytes: you can increment the high byte in order to locate the subsequent seven lines of pixels for each character space.

The Timex uses a quite straightforward method of printing characters on the screen. It has stored in ROM a table of shapes which, when

POKEd to the screen, will form the letters. We saw earlier that the routine for printing to the screen can be called at 10h; and that this jack of all trades routine also handles other channels, assuming that the correct one is selected. It has several other assets: it can move the print position when the screen is selected; it prompts you with the query "SCROLL?" if there is no room left to print in; it sets the colours; it handles the special printing modes such as OVER; and it will even expand the BASIC tokens when you send them for it to print. However, at the heart of the routine, exists code which looks up the current printing position; then it uses the ASCII character code which was passed to it in the A register when the routine was called, to find the corresponding dot pattern in the ROM. It copies the first byte of that pattern onto the screen, increments its pointer to the ROM table, and also increments the high byte of the register that holds the screen address, so that it points to the next line of pixels down the screen. The next byte of dot pattern is then copied into D-FILE 1 and the process repeated until all eight bytes are in the correct parts of screen memory. Now that the shape is in the display file, it will automatically be sent to the screen by the SCLD. The output routine also sets the attributes' byte for the character space to whatever the current colours are, and updates the print position. This is then stored in two ways: as an absolute address for the first byte of the character space; and as line and column numbers as used by the PRINT AT command. So the 'real' screen address need be calculated less frequently than it would if only the co-ordinates were stored.

Another system variable that is used by this routine is CHARS, which holds an address 256 lower than the beginning of the character shape table. For most characters you need only multiply their ASCII code by 8 and add it to CHARS; this enables you to find the address of the block of eight bytes that holds the shape of a particular character. The normal value of CHARS is 3C00h as the table is located from 3D00h to 3FFFh in the main ROM; but as CHARS is held in RAM, you can make the operating system use your own character set which is located elsewhere. If you want to create confusion, try POKE 23606,8: for further nonsense POKE 23607,0.

Much the same method is used to store another source of shapes, the *user definable graphics*, which are kept at the top of memory. When it is asked to print 1, the output routine fetches the address of the table from user definable graphics and obtains the shape from there. Before any new shapes have been defined, the RAM holds a copy of part of the ROM table, so we use normal capital letters. In addition, there is a set of *predefined graphics*, in the form of quarter character sized blocks, which are obtained by calculation. If you were to study their shapes in conjunction with their character codes, translated into *binary*, you

would probably guess how this is done. (See your manual for the codes.) The four quarters can be seen as the bits of a four bit binary number, with the prefix of 1000 binary; the shape is therefore 'made up' by a clever piece of machine code.

In order to manipulate the screen from machine code, you can use one of two approaches. You can either write your own purpose-designed printing routines, which is not as difficult as it sounds, or use the Timex's own software. If you are writing a program that does not require outstandingly fast and smooth graphics, then the available ROM routines are more than adequate. You can send all the BASIC control codes to the output routine, and even print to the bottom two lines of the screen, if you change the value of DF SZ, which holds the number of lines reserved for error messages. (Don't neglect to restore it before returning to BASIC or there will not be anywhere for the computer to print OK!) Many other useful routines are available through the function dispatcher and later I will show you how to get at these directly. For those of you intent on bettering 'Buck Rogers and the Planet of Zoom' arcade game with your own routines, then good luck, and here are two small tips.

The first relates to the border: many commercial programs for the Spectrum are marred by black flashes in the border every time the colour is changed; yet to prevent this is simple. Before changing the three low bits of port FEh to the new colour, use a HALT command so that the CPU waits for an interrupt before continuing. When it does so, this will coincide with a frame pulse, so the change of border colour will not occur during active picture time. The second tip is **Program 7**, a machine code listing which calculates screen addresses. I have only provided the OP codes and not a BASIC program to load it, because it will need to be incorporated into a machine code program as a subroutine. It can be located anywhere in free memory. To use it, load the E register with the horizontal column number as in the AT command; and load the D register with the screen pixel line number, which starts at the top with line zero and ends on 192 dec. Then call the routine. When it returns, the HL register will contain the address of the required byte in the display file, and the BC register will hold the corresponding byte of the attributes file. You may wish to shorten the program to deal only with the first byte of each character; and there is no need to use the same registers either, if these changes suit your purposes. I have kept the program simple so as to demonstrate the method more effectively: you should find the principle useful. Assuming that you have a 48K machine, perhaps you want to use the other video modes. The simplest method is to use the function dispatcher to remove the contents in Chunk 3 which will also write the appropriate bits to port FFh, thus switching the SCLD chip to the new display mode. Once

Chunk 3 is clear, you may switch between modes as much as you wish, by writing to port FFh as described earlier. Let's take each mode in turn and see how the files are arranged.

Program 7 Screen Addresses

Address	Hex Code	Assembler
SCALC	F5	PUSH AF
	7A	LD A,D
	E6F8	AND F8h
	0F	RRCA
	0F	RRCA
	0F	RRCA
	6F	LD L,A
	E618	AND 18h
	F640	OR 40h
	67	LD H,A
	7A	LD A,D
	E607	AND 07h
	84	ADD A,H
	67	LD A,L
	E607	AND 07h
	0F	RRCA
	0F	RRCA
	0F	RRCA
	83	ADD A,E
	6F	LD L,A
	7A	LD A,D
	E6C0	AND C0h
	07	RLCA
	07	RLCA
	F658	OR 58h
	47	LD B,A
	4D	LD C,L
	F1	POP AF
	C9	RET

On entry D holds the pixel line number
E holds the column number
Routine returns HL holding the display file address
BC holding the attributes file address

Display File Two follows the same pattern as the standard mode — the only problem is that you cannot use the ROM routines. The

principal benefit of this file is to facilitate instant cutting between pictures; confine the printing to D-FILE 1 and all will be fine. Here are two possible uses, both avoiding elaborate machine code printing routines. The picture is displayed from D-FILE 1; a short routine using the OP code LDIR copies it up to D-FILE 2 (you could even use BASIC if time is not important). OUT FFh,1 then forces the SCLD to display the picture now in the second file, leaving the first free for printing to by the output routine (or BASIC). When the new picture is ready then OUT FFh,0 will cause it to be displayed. The second suggestion is to store a SCREEN $ in the second file. Very elaborate graphics are often included in the title page of cassette programs: they are loaded straight into D-FILE 1 from cassette, but are lost when the screen is used by the program. By booting them up to D-FILE 2 they are still available to cut to when required.

The high res mode uses both D-FILES 1 & 2 but ignores the two attributes files. Provided you ensure that the second file, now in use as an attributes file, holds some sensible values (not black ink on black paper!) then the efforts of the output routine will be visible, as the files are mapped as for the standard mode. Indeed, a modified version of the screen address calculator can be used to find the attribute address: (use a base address of 6000h instead of 4000h). None of the colour commands of the output routine will work, but as the strength of this mode is its high colour resolution, you will need to write your own routines to really make it sing and dance.

The 64-column mode will prove the most useful for the more serious applications: wordprocessors, assemblers and the like. In this mode, the two display files each provide alternating bytes for each line of pixels, although the vertical relationships remain as for the standard mode. The output routine will print to D-FILE 1 successfully, but there will be a gap of one character in between each of the letters, so you will need a printing routine of your own to fill these gaps. Why not try dividing the column number by 2, and, if you get a remainder, adding 2000h to the result of calling the screen address calculation program?

Writing high quality graphics routines is no mean task, but you are starting off with some pretty good raw materials. You will be pleasantly surprised by some of the effects that can be achieved with just a small dose of machine code, particularly if applied to the attributes file.

CHAPTER 11
Sound

'If only it could talk', you say as you stare incomprehendingly at an error report at the bottom of the screen. In fact you may already have heard a talking Timex, as they can be persuaded to speak, albeit somewhat crudely, by software. The hardware for making sound is in two parts: the simple BEEP facility as provided on a Spectrum; and the bolt-on extra of a specialist sound effect generating chip. The beep remains, as it is the quickest way of making a noise. The clicking that occurs when the keyboard is in use comes from these beep circuits; and the signals sent to the MIC socket are from the same source. The sound chip requires more effort to get going, but it is capable of producing some spectacular effects.

The simple beep circuit is contained mainly in the SCLD. The gates responsible for decoding the CPU's attempt to write to or read from port FEh, connect to a pin of the SCLD which has some external components attached. If a write operation to port FEh sets bit four, then a high enough voltage comes out of the SCLD pin to activate the small built-in speaker: if bit three is set, then the voltage generated will be insufficient to drive the speaker, but it will still be present at the MIC socket for recording on cassette. The sharp-edged square wave that occurs when a voltage is switched between high and low, is somewhat modified by the use of a *capacitor* which rounds off the edges. The same SCLD pin is also connected to the EAR socket: by reading bit 6 of port FEh, either a digital 1 or a 0 is returned, depending on the incoming voltage. Inside the SCLD, a special kind of gate called a *Schmitt Trigger* is used to approximate the signal into a 1 or 0. If you slowly raise the voltage on the input of this kind of gate, its output will remain low until the input is around 2.5 volts, when it 'triggers' and the output goes high.

The Timex's system software creates a beep in the following manner: bit 4 of port FEh is set; a looping routine makes the CPU wait for a length of time which relates to the frequency of the beep required. When this time has elapsed, bit 4 of the port is reset, and the loop delay is repeated. One cycle of sound has now been sent to the speaker and the whole process recurs until cycles sufficient to achieve the required sound duration have been generated. There are two points to note about this routine. If we double the frequency, that is, the number of

cycles per second, by halving the delay period, then the duration of the beep will halve, unless we double the number of cycles that are generated. The beep routine calculates how many cycles of any particular frequency are needed, in order to make the sound last for the required duration. In doing this it makes extensive use of the floating point number routines. The other factor is the three low bits of port FEh which control the border colour: if you write to the port indiscriminately, without setting the bits to the right value, then the border will change. The beep routine looks up the current border colour in a system variable called BORDCR: bits 3, 4 & 5 indicate which colour is active. These bits are moved to the low three of the accumulator, so that the border remains constant when an OUT (FE),A instruction is executed.

Using the beep port from machine code threatens all the above pitfalls for the user, with one extra thrown in for good measure. A valuable advantage is the speed with which the parameter can be changed; so, for example, alternate cycles of different duration can be produced, or the shape of the cycles altered, with the low part lasting for less time than the high part. The extra pitfall I mentioned is caused by the video generating circuits that will stop the CPU when both it and the SCLD are trying to use Chunks 3 & 4 of RAM. Any machine code that is located here will run in a 'lumpy' manner, as it gets interrupted quite often. Although it is still wise to turn off the interrupts, as the ROM routine does, these cause much less 'warbling' than does the constant halting of the program, if it is in the low 16K of home RAM. If you want to create pure tones with the beep port, then you must stick with the ROM code, readily available through the function dispatcher, or use the expanded machine.

An example of what can be attained in a few bytes is given in **Program 8**. The BASIC program can simply be typed in, saved and run. The machine code listing explains what is happening and allows you to incorporate the effect in your own programs. Note how the individual tones blend into one another, an effect that you are unlikely to achieve using BASIC alone.

Program 8 Sound FX

ADDRESS	HEX CODE	ASSEMBLER
RAMTOP+1	3A485C	LD A,(5C48h)
	1F	RRA
	1F	RRA
	1F	RRA
	E607	AND 07h
	0EFF	LD C,FFh

```
2600        LD H,00h
44          LD B,H
CBE7        SET 4,A
D3FE        OUT (FEh),A
10FE        DJNZ -2
44          LD B,H
CBA7        RES 4,A
D3FE        OUT (FEh),A
10FE        DJNZ -2
CBE7        SET 4,A
D3FE        OUT (FEh),A
10FE        DJNZ -2
CBA7        RES 4,A
D3FE        OUT (FEh),A
10FE        DJNZ -2
24          INC H
0D          DEC C
20E2        JR NZ,-30
C9          RET
```

```
  10 REM        Beep port routine
  11 REM
  20 REM        Lower RAMTOP
  21 REM
  30 RESTORE : LET x=( PEEK 23730+256* P
EEK 23731)-43: CLEAR x
  40 REM
  41 REM        Poke Machine code
  42 REM        into memory
  43 REM
  50 LET x=( PEEK 23730+256* PEEK 23731)
+1
  60 FOR y=x TO x+42: READ z: POKE y,z:
NEXT y
  70 REM
  71 REM        Print USR address
  72 REM
  80 PRINT  AT 1,2;"The routine is calle
d by the"; TAB 7;"function USR ";x
  90 REM
  91 REM        Example
  92 REM
 100 PRINT  AT 12,10;"Press any key"
```

```
 110 IF  INKEY$ ="" THEN  GO TO 110
 120 LET a= USR x
 130 GO TO 110
 200 REM
 201 REM        Machine code data
 202 REM
 210 DATA 58,72,92,31,31,31,230,7,14,255
 220 DATA 38,0,68,203,231,211,254,16,254
,68
 230 DATA 203,167,211,254,16,254,203,231
,211,254
 240 DATA 16,254,203,167,211,254,16,254,
36,13
 250 DATA 32,226,201
```

Before tackling the sound chip, let's take a brief look at the way that the cassette save and load functions are accomplished by the system software. As I mentioned earlier, these routines occupy the EXROM, so direct use of them will require some manœuvring: again the function dispatcher comes to our aid. If you want to devise your own routines, perhaps to achieve a faster method of storage, then the SCLD's policy of halting the CPU, means that the best place for them would be the extra RAM area of an expanded machine. The method of saving goes along these lines:

The file name and the information as to which type of file it is, is assembled in memory as a string of bytes; to be used later as a header ie a short introductory passage; then the 'start tape' message is displayed. The machine then tests the keyboard for a press, and when one occurs, first the header, then the data itself, is formatted and sent to the MIC socket for recording. This format consists of, firstly, a leader of tone with a frequency just over 800 cycles, then one cycle of 2040 hertz. Now single bits are transmitted as a cycle of either 2040 hertz to represent a zero, or 1020 hertz for a one. The first eight bits come from the A register, which holds a flag that is set if the data to follow is header information. Next, each byte to be stored is sent, one bit at a time, as tones, to the EAR socket: after each byte the keyboard is tested to ascertain whether BREAK is being pressed, and the routine aborts if it is. At the end of the data, a final byte is written to tape: (the length of the block was held in the DE register). This byte, called the parity byte, is created by XORing each byte of data together as a checksum.

Loading information back from tape consists of the above process in reverse. The CPU monitors bit 6 of port FEh, and when it detects the leader tone, it waits for the first cycle of 2040 hertz which signals the start of data: then it reads in the bits of tape, and forms them back into

bytes. The first byte informs the CPU whether it is reading in a header or data, and the software reacts accordingly. When a suitable header is found, the information as to where the data is to be stored (PROG if it is a BASIC program), and how long it is, is read from the header information: the file name is printed to the screen and the next data file on tape is loaded into memory. At the end of the data, the parity byte from tape is compared with another which the loading routine has been building up from the incoming data: if there is a discrepancy, the routine stops with an error report. During the time that bits are read in successfully, they are also reflected in the behaviour of the border, which switches between yellow and cyan as the EAR voltage alters. You may like to set up two adjacent areas of RAM as blocks of 0s and FFhs (255), save them with a code save, then play them back into the machine. If you do, you will see the width of the stripes rolling through the border double as the FFhs are loaded.

The AY-3 8912 sound chip (PSG) is altogether different. It is designed to be a very flexible source: it can work independently of the CPU, which only needs to intervene when a change of values is required. The chip can be controlled from BASIC by the SOUND command, via the function dispatcher, or, very easily, by raw machine code. Inside the chip are a number of registers that control the output, 14 in all. In order to load any of them, two port addresses are decoded by the SCLD, to activate the input and output circuits of the sound chip. Port F5h communicates with the address lines of the chip; port F6h with its data lines. For example, OUT (F5h),7 will make register 7 of the sound chip available for writing or reading by using port F6h: OUT (F6h),0 will then load register 7 with zero. Any use of port F6h, will now access register 7, until we send a different value to the address port, F5h.

Program 9 is designed to help you experiment with the Programmable Sound Generator's (PSG) registers. It has the advantage of displaying their contents in binary as well as decimal; as, in some cases, this can be important. You can enter binary values with the BIN function in response to the 'value' prompt. The sound chip contains three tone sources and a noise source, all of which can be altered in frequency; also, an envelope shape and a cycle control. The functions of the various control registers are as follows:

Register 0–5 These are the frequency controls for the three tone channels. The channels are labelled A, B and C: each has two registers (0 and 1, 2 and 3, 4 and 5 respectively), which hold each channel's frequency. Taking channel A as an example, R0 is its fine tone and R1 its coarse: only the four low bits of R1 are used. The two registers combine to form a 12-bit number used as the period value. The PSG

requires for its operation an input clock signal that dictates its working speed: the SCLD feeds it a 1.75 megahertz clock. The sound chip counts down 16 clock cycles, and then decrements the period value. When this reaches zero the period is over, but until then the process is repeated. The tone frequency can be calculated by dividing the clock frequency (1.75 MHz), by 16 times the period value.

Program 9 PSG Register Program

```
  10 REM    PSG REGISTER PROGRAM
  11 REM
  20 REM        Set up
  21 REM
  30 GO SUB 500
  40 REM
  41 REM        Loop
  42 REM
 100 INPUT "Register?";r: IF r<0 OR r>14
 THEN  BEEP 1,10: GO TO 100
 110 PRINT  AT r+4,12; FLASH 1;r
 120 INPUT "Value?";v: IF v<0 OR v>255 T
HEN  BEEP 1,5: GO TO 110
 130 OUT 245,r: OUT 246,v: GO SUB 300
 140 PRINT  AT r+4,12;r
 150 GO TO 100
 300 REM
 301 REM        Print contents
 302 REM
 310 OUT 245,r: LET mod= IN 246: LET pow
er=128: PRINT  AT r+4,17;
 320 FOR x=0 TO 7: LET mod=mod-power: IF
 mod<0 THEN  PRINT "0";: LET mod=mod+pow
er: GO TO 340
 330 PRINT "1";
 340 LET power=power/2: NEXT x
 350 PRINT "  "; IN 246;"  ": RETURN
 500 REM
 501 REM        Print screen
 502 REM
 510 PAPER 1: BORDER 1: INK 6: CLS
 520 PRINT  TAB 2;"PSG REGISTERS"; TAB 2
0;"CONTENTS": PRINT  AT 2,0; INVERSE 1;"
Function   No."; TAB 18;"Binary   Dec. "
: PRINT
 530 FOR y=0 TO 14: READ a$: PRINT a$; A
```

```
T y+4,12:y: LET r=y: GO SUB 300: NEXT y
 540 RETURN
 600 REM
 601 REM       String data
 602 REM
 610 DATA "Ch.A Fine","   Coarse","Ch.B
Fine","   Coarse","Ch.C Fine","   Coarse
","Noise"
 620 DATA "Enable","Ch.A Ampl","Ch.B Amp
l","Ch.C Ampl","Env.Fine","   Coarse","E
nv.Shape","Port"
```

Register 6 This is the noise period register. Although by definition, noise is of a random disposition, the noise source is based on the frequency controlled by this register. The same rules apply as for the tone generators, except that only the five low bits of this register are valid, resulting in very high frequencies.

Register 7 This register is called ENABLE. It controls which of the tone sources are turned on, as it were. It also determines whether a tone channel has the noise source added to it. Bits 7 and 6 are associated with register 14: bits 5, 4 and 3, if at zero, allow noise to intermingle with the channels C, B and A respectively. Bits 2, 1 and 0 enable the tone from the same respective channels.

Registers 8–10 These are the amplitude control registers for channels A, B and C respectively. The volume of each can be set to one of 16 values, and these values are controlled by the low bits of the amplitude registers. Bit 4 is the most important: when reset to zero it causes the channel to play at the fixed amplitude set by bits 0 to 3: when set to 1, the channel's amplitude is controlled by the envelope registers.

Registers 11 and 12 These are the envelope period registers. They control the timescale with which the envelope shape or cycle complies. Both combine to make a 16-bit number, with R12 holding the most significant byte.

Register 13 This is the envelope register itself. In its simplest form, an envelope is the shape in which the amplitude of a sound dies away. Imagine banging a drum: the sound is at its loudest at first, and then gradually fades away. Envelope control simulates this by switching on the sound at its loudest level, then reducing the amplitude to zero over a duration of time determined by the envelope period register. The four low bits of the control register all have special functions. Bit 0 is the HOLD bit, and causes the formation of one envelope cycle. It has

priority over bit 3, the CONTINUE bit, which will force the envelope to repeat itself over and over again. Bit 2 can control the ATTACK; that is, the shape taken by the envelope. It is possible to reverse the normal direction, creating a crescendo which then cuts out: if you have heard a tape recording played backwards, you will be aware of the effect. Bit 1 is used in conjunction with the continue bit, and it reverses the attack mode after each cycle of the envelope: this ensures that the sound fades up and down without any sudden changes in level. If you want a one-shot effect, you trigger the envelope cycle by writing zero to the control register. To change the attack, write 00000100 binary (4 dec), and the effect will be reversed.

Register 14 This is an additional facility of the PSG that has nothing to do with sound generation at all. The chip is provided with its own I/O port, and this is the register that holds the data: as it is used by Timex to implement the joysticks, I will describe it later.

The best way to get a feel of the potential of the PSG, is to experiment: **Program 9** is designed for this purpose. However, this only allows static effects: with machine code, quick changes to the registers are possible, which facilitate the building up of even more complex noises than are possible with BASIC. Using the SOUND command you may set the PSG registers easily; but by accessing the registers directly through the IN and OUT operations, you may read what is already in them and also save time.

For the musically inclined, here are a few observations. The beep facility allows you to produce notes that relate to the musical scale: using SOUND and machine code will necessitate the calculation of your own values. Middle C is a frequency of 440 hertz; if you double the frequency of a note, the resultant sound is one octave higher. The 11 semitones in between can be calculated by knowing that the frequency of C multiplied by the 12th root of 2 will produce the frequency of C sharp; C sharp times the 12th root of 2 gives D, and so on. The twelfth root of 2 is roughly 1.059463. Happy composing!

CHAPTER 12
The Joysticks

The fact that the Timex can host a pair of joysticks is extremely welcome to games players; and the introduction of serious application programs using input devices other than the keyboard, brings joysticks into areas other than games. Ignoring the more sophisticated Track Balls and Mice, these additional input devices fall into two categories: our old friends *analogue* and *digital*. The analogue joystick can feed the computer with data that indicates exactly how far in a particular direction the stick has been moved. The Timex 2000 is fitted with the simpler digital type of joystick: the information this type supplies is limited to which direction the stick is displaced. The sockets at either end of the machine are of the standard D type connector found in many other computers and TV games. The way that the computer expects the sticks to be wired up, complies with the most common method used by other manufacturers: if you possess a pair of joysticks that work on, say, the Atari games machine, these are suitable for use with the Timex.

The principle behind this digital type of joystick is very simple. There is one input line to the stick, called the *strobe*: when the computer wishes to read the joystick, it makes this line active. When the stick is in its central position, the strobe line is not connected to anything; but when it is moved, the mechanical arrangement inside the joystick allows it to connect to four output lines, which are fed back to the computer. Push the stick up and the strobe line connects to the 'up' line; when the strobe is active, it transmits back to the computer along the up line. This is also true for the other three directions; and, in the case of 8-directional joysticks, when the stick is displaced diagonally, contact is made with two output lines. The fire button has its own line which can be bridged to the strobe when it is depressed.

From this description it is possible to guess how the Timex might go about reading the joysticks — by the use of a circuit that sends a logic one down the strobe line and connects the outputs of the stick to the data bus. In practice, this needs to be improved upon, because we need to buffer the data lines from the stick. When in a diagonal position, some joysticks are capable of shorting two lines together: this provides the Timex with the opportunity of putting the port register of the sound chip to good use. The joystick feeds data to this register: if not affected by the stick, the bits are high. The connection arrangement is as follows:

Bit 0 —	Stick up
Bit 1 —	Stick down
Bit 2 —	Stick left
Bit 3 —	Stick right
Bits 4, 5 and 6	Not connected (held high)
Bit 7 —	Fire Button

If we read I/O port A of the sound chip, its contents are transferred to the CPU; but first we must follow the steps for manipulating the PSG. The two high bits of the enable register control whether the I/O ports are able to read or write data. The Timex does not use the second port available on the PSG, so only the A port is of interest. If bit 6 of the enable register is set, then the A port is configured for output: in order to read the port we must ensure that bit 6 is not set, although this is how the operating system normally leaves it. To be absolutely certain, the machine code instructions LD A,07h ; OUT (F5h),A ; XOR A (to make A=0); OUT (F6),A ; will set the I/O port to 'read'. You would only need to do this once in a program, if at all.

The next question is how to make the strobe line active. Remember that there are two joystick ports. Although we can connect both sets of output lines from the sticks to the I/O port, if we wish to read them individually, we must control their strobe lines separately. This is achieved with the aid of the address bus. In the keyboard chapter, I explained that when the instruction IN (PORT),A was executed by the CPU, it placed the contents of the A register on the 8 high address lines. There are two simple one-transistor circuits which monitor address lines 8 and 9: when the instruction IN (F6),A is executed, if the A register was previously loaded with 1, then the circuit associated with A8 is activated, and it sends a low signal to the strobe pin of joystick socket one. The same process occurs in relation to stick two, if bit 1 of A was set prior to the IN instruction (ie A loaded with 2). Therefore, in order to read a joystick value into the A register of the CPU, we must proceed as follows: LD A, (DEh or 14 decimal) places the required register number in the accumulator; OUT (F5h),A sends the number to the PSG address port; LD A, the joystick number (1 or 2); IN A,(F6) activates the selected strobe line via the address bus, and transfers the data in the PSG's I/O port to the CPU's A register.

What of the value now read in? It is easiest to decode it in binary: each bit set to zero indicates a connection as listed earlier. For example, 01110111 would show that the fire button was depressed and the stick pulled to the right. Note that, with an eight position joystick, a value such as 11111001 would indicate that both the down and left connections were being made. The testing of individual bits is straightforward in machine code: as a more elegant and compact alternative to performing

BIT testing operations you may wish to consider rotating the A register through the carry, with codes such as RRA, and performing conditional jumps like JR NC. How you test the bits will depend on your program requirements.

I made no mention earlier of the BASIC STICK function; the values it returns are the complement of the data read in for the directions: reading the fire button requires a separate operation. If you are writing a BASIC program, then the stick function is all you need. From a machine code routine, the joysticks can be read by resorting to the function dispatcher; however, this is time consuming, so it is worth specifically creating some short code to include in any program as a subroutine.

Finally, you may want to use the joystick port for connecting a switch to indicate that some external event has occurred. Some possibilities are a microswitch that tests if a door has been opened, or pushbuttons for a quiz game. You can plug into the socket without fear of damage to the computer, as long as the device you attach is purely passive; that is, it does not have its own voltage supply. If you look at the holes on the joystick plug, with the row of five at the top, they are numbered 1–5 from right to left, and the lower row, 6–9. Pin 1 is up, 2 down, 3 left and 4 right; pin 6 is for fire and the strobe is pin 8. If you own a joystick which is of the digital type, but it won't work on your Timex, or has the wrong sort of plug, then the above information may help you to rewire it.

CHAPTER 13
The Only Safe Chunk in a Storm

If you cast your mind back to the memory map of the Timex, you may remember that Chunk 3, (6000h to 7FFFh) had special rules applied to it. The reasons why the Home Bank RAM must not be overridden by other banks for these addresses are threefold: the CPU must never lose sight of the stack; all software, wherever it is located, must have access to some method of controlling the bank switching capabilities, if any use of the additional banks (including the EXROM in the DOCK bank) is contemplated; application and language programs located in the DOCK bank or elsewhere need to use the operating routines in the system ROM. This last point is not essential, but if, for instance, a PASCAL or other language interpreter is to be written, there is no point in duplicating the routines that already exist in the two ROMs of the Timex.

The positioning of the stack has already been discussed. The bank switching routines are accessible direct; or through the function dispatcher, and this is the routine that fulfils the third requirement above. Even for the programmer who does not intend leaving the safety of the Home Bank, the function dispatcher is both of use and of interest. You need to use a SERVICE CODE to select the appropriate function; you are able to control whether it is simply jumped to, or called as a subroutine.

Let's take the case of a single jump, say to the CLEAR SCREEN function: first of all we need to know the service code. **Table 8** lists some of the most interesting of these, accompanied by a short clue as to their potential: some are obvious, others more obscure. The code for CLEAR SCREEN is 22h. In theory, no service code can be higher than a fifteen bit number, in practice they are much lower than this. We take the service code as a two-byte value (ie 16 bits) but use bit 15 as a flag to indicate if a jump or call is required. By setting the highest bit we specify a jump, and then this value is pushed onto the stack: so, for CLS, we formulate the two bytes, 8022, as the required code, say, the HL register; then PUSH HL.

All that is now needed is to call the function dispatcher at 6200h, and the task is performed for us. You may like to have a quick go at this.

Service Code	Function
00	Write a block to tape
01	Write a block from tape
02	Read a bit from tape
03	Read an edge from tape
04	General tape routine
05	Load
06	Merge
07	Save
08	Change video mode
09	Write border colour (not permanent)
0E	Get bank number
0F	Get bank number
10	Enable bank
11	Goto bank
12	Call bank
13	Transfer bytes
19	Scan keyboard
1A	Sound chip registers
1B	BEEP
1C	COPY
1D	Send a character to the screen
1E	Set print position
1F	Fix attribute byte
20	Set system attributes
21	Clear input area of screen
22	Clear screen
23	Send printer buffer to printer
24	Send single line to printer
26	NEW command
29	Select current stream
34	Flash character in A to screen
3B	Execute BASIC line
4A	Free space left
4C	PAUSE command
4D	Break key test routine
57	Screen address calculator (as PLOT)

The OP codes are LD HL,8022h ; PUSH HL ; CALL 6200h ; RET ; which in hex are 21, 22, 80, E5, CD, 00, 62, C9. Load these values into a convenient part of memory with the monitor program and run the code. You could write a few lines of BASIC to check that it actually clears the

screen: perhaps, 9900 PRINT AT 10,10 ; "A MESSAGE": RANDOMIZE USR (decimal location of code); STOP. Notice that I have provided a RET instruction at the end of the code. You may think that, as a jump has been performed, it is unnecessary; however, a return instruction at the end of the CLS routine ensures that the CPU will jump back to our code. In that case, what is the call mode of the dispatcher for? Our simple routine was in the Home Bank: when calling the function dispatcher from elsewhere, bank switching is needed, and the call mode returns control to the correct bank. Before pushing the service code onto the stack, the call routine expects us to push two bytes onto the stack, to indicate whether any values are to be passsed on via the stack. By pushing two zero values to indicate that no parameters are to be passed, and resetting bit 15 of the service code, we can use the call option; in this case unnecessary, but from other banks it is vital. The use of the stack to pass data between routines can be confusing; normally the registers suffice.

The dispatcher can provide us with other information: the addresses which it uses to locate the various functions can be examined; they can direct us to the actual routines, either for use, or to provide an insight as to how they work, and how they might be adapted for other purposes. In order to find the addresses, the procedure is as follows: the service code is stripped of bit 15 and multiplied by two, (in binary, this is simply a matter of shifting the bytes one bit to the left). This number is then subtracted from 1FFF along with the carry, which has been set. The result now points to where the address needed to find the function is stored in the EXROM. A formula for this calculation could be written as ADDRESS WHICH HOLDS FUNCTION'S ADDRESS = 1FFFh − 1 − (SERVICE CODE *2). It is tempting for me to provide the appropriate address for each function, but there is a snag: the contents of the ROMs are likely to alter as mistakes are detected and corrected by Timex. These errors may not normally manifest themselves, but when they are found, Timex will take the opportunity to correct them when ordering a batch of ROMs. This may only involve a few bytes, but in the process the locations of the routines may change. Commercial software should never (but sometimes does) call a ROM routine direct: by referencing the table in the EXROM you can be sure of finding the desired routine.

You may be puzzling how you can take a peek at the table using a monitor program, as PEEK 1FFEh gives the contents of the main ROM. One trick that does not rely on bank switching is the use of the SAVE CODE command: SAVE "EXROM" CODE 0,8192 will dump the contents of the EXROM on tape, because the tape routines themselves are contained within it, so it is enabled when they are running. You can load the code back with an offset; say, LOAD " "

CODE 32768 on the 48K machine. This would transfer the contents of the EXROM into the accessible RAM. You should find the location 1FBAh of your EXROM contains 08EAh, the start of the CLS routine in the main ROM, provided it has not been changed. Try RANDOMIZE USR for 08EAh (2282 dec), or whatever your EXROM holds. Many of the other functions require some data to be sent to them. The SET AT function (service code 1Eh), for example, will set the print position to values held in B for the line, and C for the row: so you must load BC with the desired print position before proceeding to use the dispatcher. You will need to examine carefully how the functions operate before using them.

One service code that will be of interest to 48K machine owners, is 8, which changes the video mode. It responds to a value passed in the A register: zero will change the mode to the normal one; 1 brings the second file into play, and 2, the high res mode; whilst 6 invokes the 64-column mode (with black paper). To change the paper colour, bits 3, 4 and 5 are set to a 3-bit colour value. As the additional display files require the memory space used by the stack, the function dispatcher, and the bank switching routines, they must be moved out of the way: (remember that display memory must be in Chunks 2 and 3, so that the SCLD can address it through the multiplexer.) They are relocated at the top of memory; and the stack pointer and system variables are adjusted accordingly.

Think what happens when code that includes CALL instructions is moved: unless all the ABSOLUTE addresses used to call other routines that have been moved are also changed then one big crash could ensue! To avoid this problem without having to hold two versions of the bank switching and function dispatcher routines, the EXROM contains a table of all the addresses which need to be altered. It is then a fairly simple task to add the amount the routines have moved to each absolute address affected; and, by using the same offset in reverse, return them to their old values. Thus, before the function dispatcher is 'booted up' to higher memory, it is doctored so that it will work there: when it is brought back down, repair work takes place. If you use the function dispatcher to change to a higher video mode, then don't try to call it again at 6200h in order to change back: it is now situated at FA50h. Also note that, if you wish to examine the code that performs the change: it is situated in the EXROM together with other functions with a service code of less than 0Ah.

The bank switching code that normally resides in Chunk 3 is available through the function dispatcher, but it is best to reach it by direct calls: the service codes 0Eh to 13h will point you to their location. When the system software wants to use these routines, it first checks the system variable VIDMOD: if this is non-zero, then it knows that the routine it

seeks has been moved into upper memory. The best example of simple bank switching is that which is carried out during the system set up. The bank switching routines are stored in the EXROM and need to be 'booted up' to Chunk 3 before they can be used. A short machine code program is loaded into the bottom of Chunk 3 (6000h), and then CALLED. It writes 01h to port F4h which the SCLD interprets by loading 1 into the DOCK SELECT register. This register controls which chunks of the DOCK bank are to be made active in preference over the home bank. Bits 0 to 7 of this register relate to Chunks 0 to 7 of the DOCK bank; and if the bit is set to 1, then the corresponding chunk of the DOCK bank is enabled. Therefore, by loading it with 00000001 binary, Chunk 0 is active whilst the other chunks still remain selected to the Home Bank.

The next step is to write a 1 to bit 7 of port FFh, which switches in EXROM to replace whatever is in Chunk 0 of the DOCK bank. Now all that remains to be done is to copy up to Chunk 3 (still enabled as home RAM), what the CPU sees at addresses 1000h to 1630h (the section of EXROM containing the copies of the function dispatcher and the bank switching routines). This can be accomplished by employing the very useful LDIR instruction: the DOCK register is then returned to its normal VALUE, and so is bit 7 of port FFh. Then a RET instruction places control back in the hands of the callling routine in the main ROM.

Other expansion banks need their own horizontal select mechanisms similar to the DOCK select register in order to control which chunks are active. The bank switching routines allow only one bank, be it home, DOCK or extension to have control over each chunk. Writing to the extension bank horizontal register involves using address lines as well as the output port of the sound chip, a task best left to the bank switching code provided. The presence of expansion banks and cartridges is tested for during system initialization, and they are dealt with accordingly: if a language cartridge was detected, then after the system had been set up, control would be passed to the LROS: if a device was present in an expansion bank, this would be noted in the SYSTEM CONFIGURATION and CHANNEL tables, and appropriate action taken.

The bank switching routines fall into two categories, called *indirect* and *direct*. The former perform complete tasks and use the direct routines in the course of them. There are three indirect routines. GOTO BANK allows you to pass control to any address in any bank: it needs to have four bytes pushed onto the stack to inform it where to go. First we must push the address that we wish to jump to onto the stack, with, for instance LD HL (address) ; PUSH HL. Next, the routine needs to know the bank number and a bit pattern for the horizontal selection of chunks. The pattern is the complement of what would go into the H/S

registers, so FEh will enable Chunk 0. If we LD H,(bit pattern) ; LD L,(bank number) ; PUSH HL ; we can then call GOTO BANK, and control will transfer. If you want to GOTO the Home Bank, then use FFH as the bank number: whilst the DOCK bank has the number FEh. CALL BANK is the method by which we can return to the original bank and calling routine, after the routine we have called performs a RET instruction. It needs eight bytes pushed onto the stack before being called: the same four as for GOTO BANK, followed by two 16-bit values which tell the bank switching code if any extra bytes are to be passed onto the stack. The first value is the number sent, and the second, how many are expected back. Most uses of CALL BANK will not require this function, but you will need to push two 0000hs onto the stack anyway. The final indirect routine is called XFER BYTES, short for transfer bytes. It will copy memory contents from one bank to another, and it requires the following data to be pushed onto the stack: the source bank; the destination bank; source address; destination address; number of bytes; and a flag 0081h for transferring from top to bottom, or 0001 for bottom to top (necessary if the banks overlap). The bank numbers should be formed into a 16-bit number before they are pushed onto the stack.

The last three bank switching routines are DIRECT: BANK ENABLE can bring any bank, with any chunk pattern, into play; GET BANK STATUS returns the details appertaining to the specified bank; GET NUMBER will give the specified address to whichever bank currently has control.

We have reached the point where the hardware aspects of using the Timex 2000 have been explained: how you use the information will depend on how you use your computer. Some people drive a car with no desire to open the bonnet; others prefer to have some idea of how it works even if they don't ever intend getting their hands dirty. If you see yourself as a potential computer 'mechanic', then you will need to acquire machine code skills, as well as exercising a great deal of patience. There are many books on the finer points of hardware, and Z80 machine code: as for books about patience . . . . .

This is a book for people who want to know how their Timex ticks. It will guide you, even if you have no knowledge of electronics, to an understanding of what is going on inside the case of your computer.

The book is in two sections; the first deals with the fundamental principles behind computer design. The second part takes a look at how the TS2000 series of machines are structured and, in particular, at how the screen display, keyboard and sound facilities are used.

There are both practical and helpful programs that illustrate the points discussed. You are gently introduced to the world of machine code programming with a monitor program and 'hands on' experiments.

Inside the TS2000 will take the curious beginner to the point where they can tackle those forbidding projects with a clear understanding of how their computer works.

Jeff Naylor has written several commercial computer games, including some for the TS2000. He is also a regular contributor to Popular Computing Weekly, Europe's best selling weekly computer magazine. Diane Rogers has worked mainly in theatre and television stage management. Her contribution to the partnership has been more literary than technical.

SUNSHINE

ISBN 0 946408 31 9

$11.95